End to End Machine Learning Workflow Using Automation Tools

by

Alexander Friedrich Nordin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master’s of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

Signature redacted

Author ................................................................. .................
Department of Electrical Engineering and Computer Science
May 25, 2018

Signature redacted

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

Signature redacted

Accepted by ................................................
Katrina LaCurts
Chairman, Department Committee on Graduate Theses
Abstract

We have developed an open source library named Trane and integrated it with two open source libraries to build an end-to-end machine learning workflow that can facilitate rapid development of machine learning models. The three components of this workflow are Trane, Featuretools and ATM. Trane enumerates tens of prediction problems relevant to any dataset using the meta information about the data. Furthermore, Trane generates training examples required for training machine learning models. Featuretools is an open-source software for automatically generating features from a dataset. Auto Tune Models (ATM), an open source library, performs a high throughput search over modeling options to find the best modeling technique for a problem. We show the capability of these three tools and highlight the open-source development of Trane.
Acknowledgments

Thank you to my advisor Kalyan Veeramachaneni.

A big thank you to my mother and father, Sarah and Eric Nordin, for all that they have done for me!

Thank you to my brother and sister, Bjorn and Krista Nordin. I wouldn’t be who I am today without their help, pestering and encouragement.

We acknowledge generous funding and support from CCES /KACST for this project.
Contents

1 Introduction 13
  1.1 End-to-end machine learning workflow 13

2 Automated data science 17
  2.1 Trane 18
  2.2 What’s next? 18
  2.3 Goals of this thesis 19

3 Further developing Trane 21
  3.1 Inputs to Trane 21
  3.2 Representation of prediction problems 23
  3.3 Trane algorithms 26
    3.3.1 Problem generator 27
    3.3.2 arguments generator 30
    3.3.3 cutoff_time generator 32
  3.4 Labeling 34
  3.5 Public release and community effort 35

4 The automated machine learning workflow 37
  4.1 Trane 38
    4.1.1 Trane Inputs 38
  4.2 Featuretools 43
  4.3 ATM 46
5 Demonstration of the automated workflow 49

5.1 Trane ................................. 54
  5.1.1 Outputs from Trane ...................... 57
  5.1.2 Discussion .......................... 57

5.2 Featuretools .......................... 65
  5.2.1 Discussion .......................... 67
  5.2.2 Outputs from FT ...................... 69
  5.2.3 ROC Curve Generation ................. 69

5.3 ATM .................................. 73

5.4 Conclusion ............................ 74

6 Contributions .......................... 75

6.1 Expected Contributions .................... 75

6.2 Conclusion ............................ 75

A Tables ................................ 77
List of Figures

3-1 Denormalization transforms the various datasets and their relationships into a single merged dataframe. ......................................................... 23

3-2 Filter operations remove rows from the dataframe based on some filter criterion. In this case, the criterion is “less than 10,” and 5 rows are filtered down to 2 rows. ................................................................. 23

3-3 Row operations transform the data within a row. In this case, the data is transformed to a Boolean value based on whether or not it equals the number 12. ................................................................. 24

3-4 Transformation operations operate across rows. In this case, a difference occurs across all rows and the last row is dropped, transforming 5 rows into 4. ................................................................. 24

3-5 Aggregation operations pull all the data together into a single value. In this case, the operation sums all the values. 4 rows are aggregated into a single row. ................................................................. 25

3-6 Each branch of the tree containing a filter operation, row operation, transformation operation and aggregation operation is a prediction problem. The generator enumerates problems and prunes the ones that are invalid. ................................................................. 27
3-7 Here we see entity data represented as bars across time. Each bar of data represents an entity’s data. The start of the bar indicates the first recorded observation for that entity. Similarly, the end of the bar represents the last recorded observation for that entity. The two vertical lines represent constant cutoff times - that is, cutoff times that do not vary by entity. The two vertical cuts in the data create three sections of entity data. The first section is used solely for creating features for training. The second section is used for building training labels and for generating test features. The third and final section is saved for generating test labels.

3-8 The labeller breaks the merged dataframe into individual dataframes which contain information only about a single entity. Each of these individual entity data tables are passed to the labeller, which generates labels for each entity.

3-9 The labeller executes on data by taking the individual entities’ frame data and passing it through each operation.

4-1 The inputs and outputs for each of the three systems.

4-2 The inputs and outputs to Trane.

4-3 The inputs to and output from FT. The inputs are the multiple datasets and their relationships, target entity and cutoff times. The outputs are two dataframes filled with features. One frame is filled with training features, the other with test features.

4-4 The inputs to and output from ATM. The inputs are the training features from DFS and the labels from Trane. The output is a trained model.

5-1 Histogram of the number of stars each review gives.
List of Tables

4.1 Example trips dataset. ............................................. 38
4.2 Example taxis dataset. ............................................. 39
4.3 Example merged dataset. The join keys used are taxi id and taxi company. 39
4.4 Example features generated by DFS. Trips are the entity of interest.
   Feature definitions are the column headers. ...................... 45

5.1 Information about the 7 tables in the yelp dataset. ............... 50
5.2 Selected fields of interest in the businesses table ................ 52
5.3 Selected fields of interest in the users table ..................... 52
5.4 Selected fields of interest in the reviews table ................... 53
5.5 The number of each kind of problem identified. .................... 62
5.6 The number of each kind of problem identified. .................... 64
5.7 Labels for the problem: predict how many stars the next review will
   have. Note how the cutoff times change for each entity. ............ 65
5.8 A handful of the features generated by DFS from the business entity. 69
5.9 A handful of the features generated by DFS from the users entity. 69
5.10 A handful of the features generated by DFS from the reviews entity. 70
5.11 The five problems identified by Trane as the most meaningful. .... 70
5.12 F1 and Accuracy score comparisons for basic Logistic regression models
   and ATM’s best generated model. .................................... 74
Chapter 1

Introduction

Machine learning is a field within computer science focused on giving computers the ability to learn without being explicitly programmed [2]. Machine learning can be used to help solve a wide variety of problems and has already had a massive impact on many industries, including healthcare, finance, manufacturing, retail and education. Contemporary financial traders are heavily assisted by bots that read the market, some of which outperform human traders while working with no human input. Hospitals use machine-learned programs to predict many facets of patient health: in one study, computers scanning mammograms were able to spot 52% of cancer instances as much as a year before the patients in question were officially diagnosed [5]. Marketing personalization, fraud detection, security, search, natural language processing and many more fields leverage machine learning’s potential. Soon enough, machines will be getting behind the wheel and outperforming humans on the road. Machine learning has amply demonstrated its capability to generalize and problem-solve. So how does one make machine learning work for their particular problem? In the next section we will present an end-to-end machine learning workflow.

1.1 End-to-end machine learning workflow

Machine learning can solve many types of problems, from predicting which flights will be delayed to forecasting whether Steph Curry will make his next free throw [8][9].
Although these problems vary widely, machine learning approaches each of them in a fundamentally similar manner.

The machine learning process begins with a dataset, and hinges on the collaboration between domain experts and a machine learning expert. These human experts collectively decide on a problem to solve. The machine learning expert then processes the data, creates a target variable to predict (also known as labels) and finds predictive patterns in past data (also known as features) that relate to the target variable. Finally, the expert chooses an algorithm and trains it using those labels and features. To sum up, the typical workflow looks like:

1. Organize and clean the data
2. Determine the problem
3. Generate labels based on the problem
4. Engineer and calculate features
5. Select and train a classifier or regressor

Let's consider a specific example.

Problem definition: Owen is the night shift staff manager at Massachusetts General Hospital. He's in charge of determining how many employees to schedule for each day of the week. The night shift can be variable. Some nights, there are barely any patients and many employees are idle. Other nights, every worker is rushing to and from rooms trying to help the many patients present. Owen's son won't stop bothering him about the power of machine learning to solve problems where data is available. Owen really wants to be able to allocate his resources more efficiently, so he reaches out to a machine learning expert, Daniella. Owen explains the situation to Daniella and sends her the data the hospital has collected.

Translation to machine learning problem: Daniella tries to translate Owen's needs into a concrete machine learning problem that she can solve. She spends a month and a half cleaning the data, coming up with a good question, generating labels for the machine learning task, engineering features, and training a classifier. Daniella tries to determine the number of beds that will be full on any given night. The classifier is very accurate and she expects Owen to be pleased. She presents her
work to Owen, who points out that it doesn’t account for one key detail: On busy nights, there may be more patients than there are beds. Thus, the machine learning problem doesn’t fully address Owen’s actual problem of resource allocation.

**Joint discovery of machine learning task:** Owen and Daniella go back to the drawing board to jointly generate a better question. They decide to predict the number of check-ins that will occur in a given night. Owen is excited to have this information. Daniella spends another few weeks working to build a classifier, but when they meet to go over the results, it turns out that the classifier is not accurate enough for practical use.

Although the scenario described above is hypothetical, it is representative of the field’s current state. The amount of work required by the machine learning expert, as well as the necessary interactions between her and the domain expert, mean that it typically takes months to come up with a machine learning solution to a real-world problem. The scenario highlights a few key points:

- **Lack of tools for a quick turnaround:** Each time a new question is formulated, whether jointly or individually, it takes Daniella a few weeks to produce a machine learning solution. This slow speed is the combined result of several other bottlenecks. Kaggle, a platform for hosting data science competitions, recently conducted a massive survey of over 16000 machine learning experts [3]. They found that the four largest barriers that data scientists face, in order, are a) dirty data, b) a lack of data science talent, c) a lack of management/financial support and d) a lack of a clear question to answer [6].

- **Lack of tools to ease this interaction:** Owen’s unfamiliarity with machine learning prevents him from coming up with a clear question for Daniella to answer, and makes it difficult for the two to work together.

To recap, many industries are focused on using machine learning to solve difficult problems. Automation of the machine learning process will make it simpler for industry experts to leverage machine learning’s power. Several automation tools now exist to help with the first issue “Lack of tools for a quick turnaround”. In this thesis, we develop a system called Trane to help with the second issue. We describe how existing
automated tools can work in conjunction with Trane to improve the end-to-end process. In chapter 2 we describe the goals of this thesis within the context of automated data science. In chapter 3 we describe the development of the system called Trane. In chapter 4 we present how an end-to-end workflow can be developed using the three automation tools. In chapter 5 we demonstrate the efficacy of this workflow on a real complex dataset from Yelp. Finally in chapter 6 we conclude.
Chapter 2

Automated data science

Over the past several years, researchers with MIT’s Data to AI Lab have been working to address the problems detailed in the previous chapter by developing automation tools that will (a) define problems automatically and (b) help data scientists to solve a given problem quickly. They have focused on three key areas: (1) identifying what needs to be automated and how to do it; (2) demonstrating that algorithmic automation is possible [1][4][7]; and (3) developing open-source tools and usable interfaces [4][7].

As a result, two automation tools already exist that could help Daniella once a problem has been defined:

- **Featuretools**, developed by the MIT spin-off Feature Labs, is an open source automated feature engineering tool. The tool expands the deep feature synthesis algorithm [4], making it useful for a wide variety of datasets, and includes a user-friendly interface. As of September 2017, the tool is open source for anyone to use.

- **ATM**, developed by the MIT Data to AI lab, is an open source tool for automated model selection and tuning. As of December 2017, this tool is also open source.

With the help of the two tools above, Daniella could probably reduce the time it took for her to develop, build and test her machine learning models. We make use of these two tools heavily through this thesis, and describe them in detail in later sections.

What about that initial problem, of question formulation? To address this, Schreck
and Veeramachaneni of the Data to AI lab have developed an approach called Trane [1]. Here, we describe the core principles underlying Trane and explain the contributions of this thesis.

2.1 Trane

Prediction problems are simply questions. Imagine a taxi manager who has a dataset with information about the cabs in his fleet. This dataset contains information about trips, pickup times, dropoff times, locations and fares. The taxi manager wants to know where demand will be highest for taxis in the next hour. He also wants to know where the customers in a particular area will want to travel once they’ve gotten in the cab.

Realizing that all prediction problems essentially result in a sequence of operations being applied to a dataset of entities and observations, Schreck built a language, Trane, to express them. He outlined the major ideas behind the formal language in his thesis and his paper. In his work, prediction problems are composed of 4 operation types, and there are a variety of implementations for each type. Prediction problems are generated by enumerating over the possible implementations.

Schreck proved Trane's ability to represent prediction problems using 54 problems from the data science competition website Kaggle. Although they came from a wide variety of fields, from predicting taxi trip durations to predicting sales for Walmart stores, Trane was capable of representing all 54 problems. In addition, Schreck explored restricting Trane to a smaller sub-set of operations called LittleTrane to reduce the quantity of questions generated by the generator. Even with these restrictions, 51 of 54 problems could still be expressed.

2.2 What’s next?

Trane is designed to generate prediction problem definitions given entities and observations that change over time. The entities can be any unit, such as stores, taxis,
students or patients. Observations could be trips taken in a taxi, purchases made at a store, or problems completed by a student. While Schreck’s work introduced the possibility of automatically generating prediction problems, our project tackles the next phase: How can we make this automation available for users?. There are a number of areas we could address:

- More constrained problem generation: Schreck’s Trane still generates thousands of prediction problems, too many to be useful for any human.
- Meaningfulness of problems: Many of these problems were too complex to be useful – for example, trying to predict if \( \frac{\sqrt{GlycemicIndex^2 + AverageWeight^2 + AverageWidth^2}}{PoundsConsumedperYear} \) is greater than 0.5 for a given fruit [1]. The aforementioned problem is well-defined, but it has extremely little relevance to any application we would consider meaningful.
- Usability improvements: The original Trane work was done to show proof-of-concept, and did not focus on how to make the language accessible to real users.
- Unavailability of Featuretools and ATM: When Trane was originally published, Featuretools and ATM were not publicly available. As a result, any user who read Trane could not have fully exploited its capabilities.

2.3 Goals of this thesis

Schreck and Veeramachaneni’s work provided a strong proof of concept for Trane. In this thesis, we focus on three additional steps: (1) further developing Trane; (2) developing and releasing the library as open source; and (3) connecting the three open source tools together, Trane → Featuretools → ATM.

**Further developing Trane:** To further develop Trane, we focused on the following aspects:

- We broke down the process into three distinct steps: a generator that creates prediction problem definitions, a generator that sets the arguments for the prediction problems, and a labeller that generates labels. We go over this in
further detail in Chapter 3
- We formalized Trane's inputs. Now not only can users direct the automated problem generation, they also have a very generalizable input format that will work for any dataset.
- We added an algorithm for generating cutoff time.
- We intelligently selected operations to minimize the number of meaningless problems generated.
- We defined intermediate outputs such that users can interact with the system, filter problems and thus use computing power more efficiently.

**Developing and releasing the library as open source:** One of our goals was to develop an open source, publicly usable version of Trane. To accomplish this, we started from scratch and developed an open source library. We accomplished the following things:
- An extensible "operations" class
- Tests, documentation and demos
- A well-structured repository with all the necessary information for new users to contribute.

**Connecting the three open source tools:** Finally, now that both Featuretools and ATM are open source, we connected the three libraries and developed several demos. We explain this end-to-end process by demonstrating it on a Yelp dataset in Chapter 5.
Chapter 3

Further developing Trane

In this chapter, we define the key enhancements we made to Trane. We describe the following fundamentals: (1) the format of the input data that Trane accepts, as well as of the inputs defined by the user interfacing with Trane; (2) the structure and use of prediction problems; (3) the generator; (4) the labeller; and (5) the use of receiver operating characteristic curves to intelligently select problems of interest. Finally, we provide background on our public release and explain our goals for community involvement and co-development.

3.1 Inputs to Trane

Input data: Trane is designed to generate prediction problems for datasets with multiple entities and multiple tables. Thus, input data for Trane contains:

- Multiple csv files
- Metadata: Trane requires a metatable that provides information about the input dataset and specifies the types of each column. Generating this metatable is straightforward, but requires some basic knowledge of Java Script Object Notation (JSON) along with attention to proper structure. The generator uses the metatable to avoid generating questions that are meaningless or unexecutable by assessing which operations each column of data will and won’t be compatible with. For instance, imagine trying to take the arithmetic sum of a column that
is filled with text entries like “The restaurant service was mediocre”, “I really liked the sweet potato fries” and “terrible service never coming back.” It is impossible, as this particular operation can’t be performed on this particular data. The information in the metatable allows us to avoid that issue, reducing the number of meaningless problems generated and improving the fidelity of generated problems.

**Denormalization**: Given csv-formatted data tables and a `meta.json` that details relationships between the tables, Trane first denormalizes everything into a single merged table containing all of the information. Each row of this merged table or dataframe represents a unique entry and must have a unique time associated with it. Tables 4.1, 4.2, and 4.3 on page 38 provide examples of denormalization.

**User input**: To generate a prediction problem, the user specifies the following inputs:

1. **Relationships**: The user specifies the relationships between the tables. For instance, there is a one-to-many relationship between the bosses.csv file and the employees.csv file.

2. **entity id column**: The entity id column specifies the entity a user is interested in predicting information about. For instance, a taxi manager’s chosen entity id column would be taxi_id because the manager wants to predict things about the taxis in his fleet.

3. **label generating column**: The label generating column is of particular interest to the user and typically contains the quantity they are most interested in. For instance, a taxi driver would likely want to generate questions about trip fares, because his goal is to maximize profit.

4. **time-index column**: The time-index column contains timestamps for all of the entries in the dataset. This is vital for segmenting the data according to cutoff times later in the process.

5. **filter column**: The filter column contains information that a user wants to perform a filter operation on. For instance, the filter column could specify each taxi’s operating district, allowing users to filter taxis by the locations they typically operate in.
Note: It is simple (and often useful) to change the entity id or label generating columns in order to construct additional and novel problems.

![Figure 3-1: Denormalization transforms the various datasets and their relationships into a single merged dataframe.](image)

### 3.2 Representation of prediction problems

A prediction problem is composed of four operations that take place in a specific order: filter operation; row operation; transformation operation; aggregation operation. These operations are meant to be applied to a slice of data pertaining to an entity-instance, and generate a training example specified by the id of the entity-instance, the time point at which the prediction is desired, and the target value that needs to be predicted. Consider an example prediction problem: “How much money will this taxi make tomorrow?”. Let’s say the time point at which the prediction is desired is April 1, 2016. (Later, we will explain how these desired time points are generated).

For a particular taxi – let’s say id = 4353 – we take the data between April 1st and April 2nd, and apply operations that will generate the target variable and predict the number of trips. We define the four operations below:

![Figure 3-2: Filter operations remove rows from the dataframe based on some filter criterion. In this case, the criterion is “less than 10,” and 5 rows are filtered down to 2 rows.](image)
**Filter Operation:** Filter operations allow the prediction problem to selectively prune data during its execution. They operate over the dataset, applying some decision criterion to data within the filter column in order to dictate which rows are retained and which rows are eliminated. For instance, if a filter operation is applied that only accepts values that are less than 10, rows with values greater than or equal to 10 are pruned from the data.

Two examples of filter operations are less-than and equals, both of which are self-explanatory. Both operations require arguments to perform their comparisons; that is, they need a value to perform their comparison against. For instance, the equals operation paired with argument 12 checks if the value is equal to 12.

![Filter Operation Example](image)

Figure 3-3: Row operations transform the data within a row. In this case, the data is transformed to a Boolean value based on whether or not it equals the number 12.

**Row Operation:** A row operation takes in all of the data in the label generating column, applies its function to every row in that column, and returns the modified value in a new column. For instance, a label generating column that contains the values 5, 12 and 17, the row operation < and a threshold of 4 will produce the output: False, False, False. Some examples of row operations are > and ==. Both of these operations also require set arguments in order to perform their comparisons.

![Row Operation Example](image)

Figure 3-4: Transformation operations operate across rows. In this case, a difference occurs across all rows and the last row is dropped, transforming 5 rows into 4.

**Transformation Operation:** A transformation operation works across rows in the data table. It takes in the label generating column and returns a column with at
least 1 fewer row, thereby helping the data converge towards a single answer. (The exception is the identity transformation operation, which simply returns the column as is.)

Some example transformation operations are \texttt{diff} and \texttt{object-frequency}. \texttt{diff} calculates the difference between two consecutive rows. Figure 3-4 shows an example of this operation in action. \texttt{object-frequency} transforms each value in the column into how frequently that object appears in the data. The objects are sorted and their frequencies are placed into the label generating column in order of their sorted positions. For instance, if the output True is seen 16 times and the output False is seen 12 times, the column would contain the values 12, 16 in that order. (Note that when sorted, False comes before True in Python).

\begin{center}
\begin{tabular}{ |c|c| }
  \hline
  label generating column & Other columns... \\
  \hline
  784 & ... \\
  329 & ... \\
  561 & ... \\
  622 & ... \\
  \hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ |c|c| }
  \hline
  label generating column & Other columns... \\
  \hline
  2296 & ... \\
  \hline
\end{tabular}
\end{center}

Figure 3-5: Aggregation operations pull all the data together into a single value. In this case, the operation sums all the values. 4 rows are aggregated into a single row.

\textbf{Aggregation Operation}: Aggregation operations pull all the remaining rows in the column in order to form a single answer. They are only slightly different from transformation operations. Aggregation operations take in an entire data frame and return just a single value.

Some example aggregation operations are \texttt{sum} and \texttt{first}. \texttt{Sum} adds up the rows in the label generating column. \texttt{First} selects the first row’s data from the label generating column.

\textbf{Our example problem}: Let’s go back to our example problem, “How many trips will taxi \texttt{id = 4353} take over the next day, starting on April 1st, 2016”? Here the filter column is the time-index column which includes the \texttt{timestamp} for every trip. The label generating column is the \texttt{fare} column, and operations are:

- Filter operation: \texttt{timestamp < 04:03:2016:00:00:00}
- Row: Identity
- Transformation: Identity
- Aggregation: sum

The data slice corresponding to taxi id = 4353 after the time point April 1, 2016 is run through this set of operations, which removes the data beyond April 2, 2016, applies sum to the label generating column fare, and generates the target value. Let’s say this sum is $654 At the end of the execution, one obtains the training example < taxi.id = 4353, cut off.time = April 1, 2016, label = 654 >.

In summary, to fully generate training examples, one needs:

- a sequence of operations that can be applied to a data slice for an entity-instance;
- a timepoint at which the prediction is desired (the cutoff.time); and
- the arguments for the operations as needed.

In the next section, we describe the three algorithms we use to generate these three requirements. First, the generator generates the sequence of operations that define a prediction problem. The cutoff.time time generator algorithm generates the cutoff.time times for each entity-instance, and the third algorithm generates the arguments for each operation in the sequence.

### 3.3 Trane algorithms

The next three sections describe the algorithms we have designed. Before we begin, we will define our notations here.

*Raw Inputs:* The inputs to Trane include a set of data frames and the four aforementioned columns of interest.

- **Merged table** \( T \). The merged table is the result of denormalizing all the input tables. In other words, \( T \) contains all the relevant information from every input dataset.
- **entity id column** \( C_{id} \) is a column containing the id’s of the entity-instances. For each entity-instance defined in \( C_{id} \), Trane will generate a label and a cutoff.time.
- **filter column** \( C_{f} \) Filtering is be performed using the data in this column.
• **label generating column** $C_l$ This column contains the information we aim to make predictions about.
• **time-index column** $C_t$ shows timestamps.

*Notations for operations* Each prediction problem in Trane is made up of 4 operations $O = \{o_1, o_2, o_3, o_4\}$.

- $o_1$ is always a filter operation. The output of $o_1$ is a $0/1$ vector.
- An operation $o_i$ may need some arguments given by $a_i$.
- $o_i(C, a_i)$ applies operation $o_i$ using $a_i$ on column $C$.

### 3.3.1 Problem generator

![Diagram](image)

Figure 3-6: Each branch of the tree containing a filter operation, row operation, transformation operation and aggregation operation is a prediction problem. The generator enumerates problems and prunes the ones that are invalid.

A generator looks to chain sequences together as follows: Filter Operation $\Rightarrow$ Row Operation $\Rightarrow$ Transformation Operation $\Rightarrow$ Aggregation Operation. Every possible problem is traversed, and if the problem is not compatible with the input data, it is immediately pruned.

The generator uses the meta information from meta.json to chain operations together in specific ways.

**Type checking at generation:** The meta.json file informs our type checking and ensures the validity of the prediction problem. Additionally, in our implementation, all potential input and output types are specified for every operation. Given a filter and label generating column, their types, the operations in a prediction problem, and the input/output specifications of each operation, we know how the data will be
transformed as it moves through the pipeline of a prediction problem.

To begin with, imagine we pick a filter column that is of type "integer." That means only filter operations that are capable of using an integer as input will be selected. Now consider the row operation implementation "less than." The operation will transform the input data into a Boolean value, either True or False. The operation after a row operation is a transformation. In this case, the transformation operation implementation selected must be able to accept a Boolean as input, because that is the output of the row operation. This static checking occurs before any problems are executed. During the execution of the prediction problem, a run-time check ensures the data matches the expected type found in the metatable information.

**Output of the generator**: Once a generator is run, it produces the following outputs.

- Prediction problem definitions in a JSON file. Here's an example of a prediction problem in its JSON format.

```json
{
  "prediction_problems": [
    {
      "operations": [
        {
          "OpType": "FilterOpBase",
          "SubopType": "AllFilterOp",
          "column_name": "user_id",
          "iotype": [
            "text",
            "text"
          ],
          "hyper_parameter_settings": {}
        },
        {
          "OpType": "RowOpBase",
          "SubopType": "IdentityRowOp",
          "column_name": "stars",
          "iotype": [
            "integer",
            "integer"
          ],
          "hyper_parameter_settings": {}
        },
        {
          "OpType": "TransformationOpBase",
          "SubopType": "IdentityTransformationOp",
          "column_name": "stars",
          "iotype": [
            "integer",
            "integer"
          ],
          "hyper_parameter_settings": {}
        }
      ]
    }
  
```
Listing 3.1: A prediction problem translated into a JSON format. A user may open the JSON code and change certain values, such as hyperparameter settings. The input to the labeller is a JSON file, thus the user can simply pass the modified problems into the labeler. The hyperparameter settings below refer to arguments for each of the operations.

- Natural language translation: Translating problems from Trane’s underlying language – the sequence of operations in the specified order – into English greatly reduces the barrier to understanding and using Trane. We use a simple rule-based approach to generate these sentences. Each operation has a fixed translation into English. The operation’s translations are programatically combined with proper English syntax to generate flexible and comprehensible English definitions of underlying prediction problems.

For instance, consider a problem that involves a dataset of star-based business reviews. 1 star denotes a terrible experience, while 5 stars denotes a terrific one. Trane may generate the following problem: “Predict if the next review will be less than 5 stars.” While we generated that translation by hand, the translation Trane would provide is as follows: “For each business, predict whether the op stars is less than 5, after 80% of the entitys data has elapsed”. The part of the sentence referring to the elapsed data has to do with cutoff.time, as the training cutoff times in this example are specified up to 80% of the data, and so the next review that would be predicted is the review just after 80% of the
other reviews. Future work could be done to improve these methods or even search for the best possible ones.

**Tweaking prediction problem definitions**: We provide flexibility regarding the generated prediction problem’s structure. Users may inspect and tweak prediction problem operations and their arguments. Advanced users can create arbitrary prediction problems by modifying the problems in the saved prediction problems JSON file. For instance, they can create something using two filter operations, three transformation operations, a row operation and finally an aggregation operation. Essentially, we provide a sandbox for users to easily define prediction problems. Future work could be done to build a user interface that allows users with no technical knowledge to create prediction problems by simply dragging and dropping items, setting arguments, and using Trane to execute and label.

### 3.3.2 arguments generator

For a prediction problem, we expect the user to specify arguments. However, in the spirit of full automation and enabling exploration, we define two approaches for automatically generating arguments. The first approach generates arguments for the filter operation. The first algorithm searches among unique values within the data to select the value that filters a fraction of data closest to a pre-specified amount. (This is set to 80% in the current implementation.) The second approach generates arguments for all the other operations. It seeks to choose argument settings that produce a highly varied output, in order to have maximum entropy once the column is operated on. The algorithms execute on all of the merged data.
Algorithm 1 arguments generation

Brief reminder of the notation:

\( T \) merged dataframe
\( O \) four distinct operations that form a prediction problem
\( C_{id} \) entity id column
\( C_l \) label generating column
\( C_f \) filter column
\( C_t \) time-index column

1: procedure \textsc{GenerateArguments}(\( T, O, C_{id}, C_l, C_f, C_t \))
2: \( V_f \leftarrow \text{UNIQUE}(C_f) \) \Comment{Get all unique values in filter column.}
3: \( a_i \leftarrow \text{SelectByRemaining}(80\%, V_f, C_f, o_1) \) \Comment{Leave 80% data.}
4: \( T \leftarrow o_1(T, a_1) \)
5: \( C_2 \leftarrow C_l \)
6: for \( i \leftarrow 2, 3, 4 \) do
7: \( V_i \leftarrow \text{UNIQUE}(C_i) \) \Comment{Get all unique values in label column.}
8: \( a_i \leftarrow \text{SelectByDiversity}(V_i, C_i, o_i) \) \Comment{Find threshold with best diversity.}
9: \( C_{i+1} \leftarrow o_i(C_i, a_i) \)
10: end for
11: return \( a_1, a_2, a_3, a_4 \)
12: end procedure

13: procedure \textsc{SelectByRemaining}(\( x, V, C, o \))
14: \( \text{best} \leftarrow 1 \)
15: \( \text{best}_v \leftarrow 0 \)
16: for \( v \leftarrow V \) do
17: \( x' \leftarrow \frac{\sum a(C, v)}{|C|} \)
18: if \( |x' - x| < \text{best} \) then
19: \( \text{best} \leftarrow |x' - x| \)
20: \( \text{best}_v \leftarrow v \)
21: end if
22: end for
23: return \( \text{best}_v \)
24: end procedure

25: procedure \textsc{SelectByDiversity}(\( V, C, o \))
26: \( \text{best} \leftarrow 0 \)
27: \( \text{best}_v \leftarrow 0 \)
28: for \( v \leftarrow V \) do
29: \( x \leftarrow \text{entropy}(o(C, v)) \)
30: if \( x > \text{best} \) then
31: \( \text{best} \leftarrow x \)
32: \( \text{best}_v \leftarrow v \)
33: end if
34: end for
35: end procedure
3.3.3 cutoff_time generator

Figure 3-7: Here we see entity data represented as bars across time. Each bar of data represents an entity's data. The start of the bar indicates the first recorded observation for that entity. Similarly, the end of the bar represents the last recorded observation for that entity. The two vertical lines represent constant cutoff_times—that is, cutoff_times that do not vary by entity. The two vertical cuts in the data create three sections of entity data. The first section is used solely for creating features for training. The second section is used for building training labels and for generating test features. The third and final section is saved for generating test labels.

In Trane, we try to predict what will happen to entities in the future. Cutoff times are used to prevent information from leaking across training boundaries. Such leakage reduces the fidelity of training and testing metrics and probably hurts the model's true predictive power. We are concerned with eliminating information leakage that would provide unfair insight to the training or testing features, allowing for a perfect or unfairly performant prediction. Consider the problem of predicting when a student will drop out from a massive open online course (MOOC). Once the student has dropped out, the answer to the question is clear. If the student drops out before the first cutoff time, the answer is obvious and provided to the training segment of the data, hence leaking vital information. We seek to prevent that kind of information leakage in order to maintain the fidelity of the predictive models and their accuracy metrics.

Cutoff times segment the data into three distinct sections. The first section is the feature training section. This data is used solely for generating features for the training of the predictive model. The second section is the training label and test
features section. This data is used for generating labels corresponding to the training features and for generating features used in the test of the predictive model. The third section is the test label section. This data is used solely for generating labels for testing the predictive model. There are two cutoff times, the training cutoff time and the label cutoff time. The training cutoff time segments sections 1 and 2. The label cutoff time segments sections 2 and 3.

Entities may have unique start and end times. Consider stores: A recently opened store won’t have nearly as much historical data as the store one town over that has been open for 20 years. We seek to select cutoff times for entities that prevent any information leakage. We may either select two constant cutoff times for all entities or dynamically assign cutoff times based on entity-specific information. In summary, cutoff times prevent vital information leakage and segment the data into sections that can be used for training and testing a predictive model.

**Dynamic cutoff-time generation algorithm:** Generating cutoff times properly requires an intimate knowledge of all of the data. This could be gained by inspecting each entity’s data in the context of every prediction problem generated, but having the user inspect hundreds of entities’ worth of data for every problem generated would add up to many hours of work and defeat the purpose of the system. Thus, we define an algorithm for generating cutoff times. The algorithm uses the information about time in each entity to determine values for the two required cutoff times. To put it simply, the algorithm generates cutoff times by segmenting the data at times which provide roughly 60% of the data for generating training features, 20% for generating testing features and training labels, and 20% for testing labels. Note that these percentages may be updated by the user, as we show in the MLF demo in chapter 6.
Algorithm 2 Dynamic cutoff.time generation

Brief reminder of the notation:

T merged dataframe

O four distinct operations that form a prediction problem

C_{id} entity id column

C_l label generating column

C_f filter column

C_t time-index column

1: procedure GENERATE_DYNAMIC_CUTOFF_TIME(T, O, C_{id}, C_l, C_f, C_t)
2: \hspace{1em} V_d \leftarrow \text{UNIQUE}(T_{C_{id}})
3: \hspace{1em} \text{for } v \leftarrow V_d \text{ do}
4: \hspace{2em} t_v \leftarrow \text{SORT}(T_{C_t}^{(v)})
5: \hspace{2em} n_v \leftarrow |t_v|
6: \hspace{2em} t_{train}[v] \leftarrow t_v[\lfloor 0.6n \rfloor]
7: \hspace{2em} t_{test}[v] \leftarrow t_v[\lfloor 0.8n \rfloor]
8: \hspace{1em} \text{end for}
9: return t_{train}, t_{test}
10: end procedure

3.4 Labeling

Figure 3-8: The labeller breaks the merged dataframe into individual dataframes which contain information only about a single entity. Each of these individual entity data tables are passed to the labeller, which generates labels for each entity.

Once we have managed to specify the prediction problem as a set of operations O, the arguments for each operation specified by a_1, a_2, a_3, a_4 and the cutoff.times, t_{train} and t_{test}, we are ready to label entities in the data. The output of the labeller is the set of training and testing examples. To create training examples, we provide the t_{train} to the labeller and follow these steps:

- Extract an entity’s data from the merged data frame T.
- Segregate the data based on the cutoff_time.
- The remaining data is passed to the first operation. The output of the first operation is then fed to the second operation, and so on and so forth until we've executed all the operations in a prediction problem, generating the target value or label.
- Check the type of the data against the expected type.
- Output < entity_id, cutoff_time, label >
- Repeat from steps 1 - 5 for each entity in the data.

We can repeat the above procedure to generate testing examples.

Figure 3-9: The labeller executes on data by taking the individual entities’ frame data and passing it through each operation.

3.5 Public release and community effort

We completely rebuilt and redesigned Trane from the ground up. We attempted to develop a well-structured, well-documented code with many unit tests for ensuring stability. Furthermore, we designed the system to be modular. We have open-sourced the software, which is available at https://github.com/HDI-Project/Trane. We hope that the community of practitioners and machine learning experts will help to contribute new operations, arguments set algorithms, or cutoff_time generate algorithms. Users may simply create pull requests with the new operations they would like to include. Instructions are available within the code for how to create new operations.
Chapter 4

The automated machine learning workflow

Three software packages – Trane, Featuretools and ATM – are combined in that order to form an end-to-end machine learning problem generation and solving system. All three of these packages are open source and are publicly available on Github 1 2 3. Each is a system designed to execute on certain inputs and provide certain useful outputs for the next step in the machine learning workflow. Data and meta-data is sent into Trane for generating prediction problems, and then to labels to generate training and testing examples. Featuretools then generates the features automatically. Finally, ATM uses the labels and features to generate optimized models. In this chapter we will describe this end-to-end workflow.

Figure 4-1: The inputs and outputs for each of the three systems.

1Trane https://github.com/HDI-Project/Trane
2Featuretools https://github.com/featuretools/featuretools/
3ATM https://github.com/HDI-Project/ATM
4.1 Trane

The system, Trane, is designed to generate prediction problems using metainformation about data. Trane works on data that contains entities and observations about those entities over time. Entities can be any unit or object (a taxi, for instance). The observations across time can take many forms, including taxi trips.

Prediction problems are questions that can be asked about a dataset. For instance, a prediction problem might be be: "how many trips will a specific taxi go on tomorrow?" Trane generates prediction problems on the scale of tens to hundreds.

4.1.1 Trane Inputs

Trane requires four inputs. We will first enumerate them, then describe them in depth.

1. Data tables: $T_1, T_2 \ldots T_n$
2. Four Columns of Interest: $C_t, C_e, C_t, C_e$
3. Meta Information: $M$
4. Relationships: $R_1, R_2 \ldots R_{n-1}$

**Data tables**: Data is provided in raw tabular form. Joining relationships must be specified among the tables such that the disjoint tables may be successfully merged into a single table. For instance, two tables, each with a column containing keys to look up the data in a row, may be joined on their key columns to produce a new table containing all of the information originally contained in each disjoint table. Tables 4.1 and 4.2 are two disjoint tables. Table 4.3 contains the merged table.

Table 4.1: Example trips dataset.

<table>
<thead>
<tr>
<th>trip id</th>
<th>fare</th>
<th>distance</th>
<th>taxi id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7.90</td>
<td>3.2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4.50</td>
<td>1.6</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>13.25</td>
<td>4.2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>21.40</td>
<td>7.0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4.65</td>
<td>1.5</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>12.75</td>
<td>4.5</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 4.2: Example taxis dataset.

<table>
<thead>
<tr>
<th>taxi number</th>
<th>seats</th>
<th>taxi company</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>Beck</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>TaxiCo</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>TaxiCo</td>
</tr>
</tbody>
</table>

Table 4.3: Example merged dataset. The join keys used are taxi id and taxi company.

<table>
<thead>
<tr>
<th>trip id</th>
<th>fare</th>
<th>distance</th>
<th>taxi id</th>
<th>seats</th>
<th>taxi company</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7.90</td>
<td>3.2</td>
<td>0</td>
<td>6</td>
<td>Beck</td>
</tr>
<tr>
<td>1</td>
<td>4.50</td>
<td>1.6</td>
<td>1</td>
<td>4</td>
<td>TaxiCo</td>
</tr>
<tr>
<td>2</td>
<td>13.25</td>
<td>4.2</td>
<td>1</td>
<td>4</td>
<td>TaxiCo</td>
</tr>
<tr>
<td>3</td>
<td>21.40</td>
<td>7.0</td>
<td>0</td>
<td>6</td>
<td>Beck</td>
</tr>
<tr>
<td>4</td>
<td>4.65</td>
<td>1.5</td>
<td>0</td>
<td>6</td>
<td>Beck</td>
</tr>
<tr>
<td>5</td>
<td>12.75</td>
<td>4.5</td>
<td>2</td>
<td>4</td>
<td>TaxiCo</td>
</tr>
</tbody>
</table>

4 Columns of Interest: There are four columns that a user must specify in the input to Trane.

1. label generating column
2. time-index column
3. entity id column
4. filter column

The user specifies each column using the column’s name.

The label generating column typically contains the quantity the user is most interested in making predictions about – for instance, a taxi driver would likely want to generate questions about the fare of a trip.

The time column contains timestamps for all of the entries in the dataset – for instance, the taxi trip start times.

The entity id column contains the entities a user is interested in making predictions about – for instance, a taxi manager’s chosen entity id column might be the taxi id column.

The filter column contains information that a user wants to perform a filter operation on. For instance, if the user wants to filter the taxis according to preferred
location, this column could be the taxis' operating districts.

It is simple and can be useful to change the entity id column or label generating column in order to generate new questions about the data – for instance, changing the entity id column from taxi id to vendor id in order to predict characteristics of vendor activity.

We chose to allow the user to specify the label generating column, the filter column and the entity id column instead of trying every possible combination of these columns. The latter strategy would generate tens of thousands of problems, many of which would be irrelevant.

**meta.json** We require a file containing metainformation about the datasets written in Java Script Object Notation (JSON). The metainformation may be thought of as another table that specifies the types of each column. The data belongs to fixed types and subtypes, which are defined in our implementation. There are five types defined by Trane: categorical, text, number, datetime, and id. The categorical type has three subtypes: categorical, Boolean and ordered. The number type has two subtypes: integer and float. An example meta.json is shown below.

```json
{
  "tables": [
    {
      "fields": [
        {
          "name": "vendor_id",
          "type": "number",
          "subtype": "integer"
        },
        {
          "name": "taxi_id",
          "type": "id"
        },
        {
          "name": "trip_id",
          "type": "number",
          "subtype": "integer"
        },
        {
          "name": "distance",
          "type": "number",
          "subtype": "float"
        },
        {
          "name": "duration",
          "type": "number",
          "subtype": "float"
        },
        {
          "name": "fare",
          "type": "number",
          "subtype": "float"
        },
        {
          "name": "num_passengers",
          "type": "number",
          "subtype": "integer"
        },
        {
          "name": "start_time",
          "type": "datetime"
        },
        {
          "name": "end_time",
          "type": "datetime"
        },
        {
          "name": "unique_entry_id",
          "type": "number",
          "subtype": "integer"
        }
      ]
    }
  ]
}
```

**Relationships:** The relationships input contains information detailing how to merge the data tables. Specifically, it contains the two file names and the join keys to be used for performing a merge between them. Relationships are defined as a list of tuples.
The list contains \( n-1 \) entries where \( n \) is the number of input tables. The first entry in the tuple is the first filename, the contents of which are used to generate the first table. The second entry is the column containing the join key for the first table. The third entry is the second filename, the contents of which are to be used to generate the second table. The fourth and final entry is the name of the column containing the join key for the second table. The specific type of relationship, for instance, one-to-one or one-to-many, does not need to be specified. Below, is an example code for the relationships input between tables 4.1 and 4.2 on page 38. Assume the data in the taxis table came from taxis.csv and the data from the trips table came from trips.csv.

\[
\text{relationships} = (\{'taxis.csv', 'taxi number', 'trips.csv', 'taxi id'\})
\]

Figure 4-2: The inputs and outputs to Trane.

**Trane execution and outputs:** The prediction problem is the backbone of Trane’s functionality. A prediction problem is composed of four operations. Operations are functions that transform a dataframe into some new dataframe. Operations belong to one of four classes. Filter operations filter data. Row operations transform data within a row and return a dataframe of the same dimensions. Transformation operations transform data across rows returning a new dataset with fewer rows. Aggregation operations accumulate the dataframe into a single row. Prediction problems are composed of one of each of the 4 types of operation in a specific order: filter operation to row operation to transformation operation to aggregation operation.
Prediction problems are generated via a cartesian product over all possible operations that fit the order specified above. More detail about this process and operations is available in chapter 3. After generation, the problems are validated to ensure they can be executed on the given data. For instance, imagine trying to take the sum of a column that is filled with strings. The arithmetic sum of "the taxi drove too slow", "the driver was very friendly" and "he drove fast!" means nothing. Invalid problems are discarded.

Some operations require hyper parameters. For instance, a greater than row operation requires a threshold value to perform the comparison against. Trane automatically generates parameters for operations within each prediction problem using algorithms we built. The algorithms are defined in chapter 3.

The problems are then written to a JSON file. The user can modify the file to change the order of operations or tune the parameters to their pleasing. For instance, a user could change the threshold for a greater row operation from 2.0 to 50.0. Additionally, a user could add an equals row operation to the end of a problem to transform a numerical output to a boolean value, thereby changing the regression problem to a classification problem.

Cutoff times are two specific times in the dataset where data should be segmented. The first cutoff time, training cutoff time, divides the portion of data used for generating training features and the portion of data used for generating training labels and testing features. The second cutoff time, testing cutoff time, divides the portion of data used for generating testing features and training labels from the portion of data used for generating testing labels. Cutoff times may be automatically generated by Trane according to algorithms defined in the Key Ideas chapter, or specified by the user. It is crucial that enough data is allocated to each of the three portions of data.

The problems are executed operation by operation on each entity’s data to generate labels. The data is segmented at the testing cutoff time. Data prior to the cutoff time is used to generate training labels. All of the data, including that after the testing cutoff time, is used to calculate testing labels. The output is a list of tables, one table for each problem. Each table contains 4 columns. The first column has the entity.
The second column has the training label. The third column has the testing label. The final column contains the testing cutoff time.

There are four outputs from Trane’s execution, enumerated below.

1. Prediction Problems
3. Training and Testing cutoff times for each entity.
4. Training and Testing Labels: $L_{train}, L_{test}$

The training and testing labels are percolated through the rest of MLF.

## 4.2 Featuretools

Given training and testing examples, Featuretools can automatically generate features. It takes 4 inputs. we will first enumerate them, then describe them in depth. Note, the inputs are very similar to the inputs given to Trane.

1. Data tables: $T_1, T_2, ... T_n$
2. Relationships: $R_1, R_2, ... R_{n-1}$
3. Target entity: $C_{id}$
4. cutoff times
The inputs are the multiple datasets and their relationships, target entity and cutoff times.

The outputs are two dataframes filled with features. One frame is filled with training features, the other with test features.

*Data tables* are input as Pandas dataframes and are simply the dataframe representation of the tables originally passed in to *Trane*.

*Relationships* define the relations between different dataframes and are used by *Featuretools* to generate deep features. The structure of the relationships input to *Featuretools* is the same as we passed for *Trane*. However, the type of relationship between each table must be specified. The first dataframe in the relationships tuple represents the parent entity and the second dataframe represents the child entity. The parent entity typically has many children. For instance, a taxi entity has many trips. Instead of filenames, like *Trane* relationships expect; relationships for *Featuretools* expect dataframe names.

*Target entity* is the entity for which we want to make features. Recall the entity id column input to *Trane*; the target entity provided to *Featuretools* is the same.

*cutoff times* is a pandas dataframe mapping each labeled example to its cutoff time. The cutoff times prevent any data made available past the cutoff time in an entity from being available to the feature generation algorithm while generating features.
Feature tools execution and output: Featuretools operates over the inputs to generate features most relevant to predicting characteristics of the specified entity. For instance, if the entity was taxi id, Featuretools would select the most relevant features for predicting characteristics of taxi’s behavior, such as fare. Featuretools’s algorithm, Deep Feature Synthesis (DFS), follows relationships in the data to a base field, and then sequentially applies mathematical functions along that path to create the final feature [4].

We run the DFS algorithm twice. To generate features for training examples, we run the algorithm with the cutoff_times set as the training cutoff_time. To generate features for testing examples, we run the algorithm with the cutoff_times set as the testing cutoff_time.

Featuretools generates two pandas dataframes. One contains features for training examples, and the other contains features for testing examples. Finally, we one-hot encode the features. Example features are shown in figure 4.4.
4.3 ATM

Given features for training examples and their corresponding labels, ATM searches through the model space, selects models and tunes their hyperparameters.

1. Joint Table: \( J \) (simply a concatenation of \( F_{\text{train}} \) and \( L_{\text{train}} \))

ATM takes a single input, the joint table. Each row contains all the features and the last column contains the label. To create the input, we merge the training features generated by Featuretools with the training labels built by Trane.

**ATM output:** We select the model with the best performance metrics to use on the testing features and testing labels. We can now feed in our test features to determine the accuracy of our model. Additionally, we can produce ROC curves for the model.

Alternatively, we may simply train Logistic Regressions classifiers. The basic classifiers provides a good baseline, runs significantly faster than ATM and still allows for direct comparison of AUC scores obtained by various problem definitions. We may then run the slow, yet powerful ATM tool on the problems we determine to be most meaningful.

1. Models: \( model_1, model_2...model_k \)

![Figure 4-4: The inputs to and output from ATM. The inputs are the training features from DFS and the labels from Trane. The output is a trained model.](image)

Finally, using either the ATM or Logistic Regression model, we can generate ROC
curves to assess the AUC scores of each problem. The problem's with the lowest AUC scores are most likely to be meaningful, given their difficult to predict nature.
Chapter 5

Demonstration of the automated workflow

In this chapter, we show how a Machine Learning Friend (MLF) executes on a set of “Yelp review data” [10]. The MLF is composed of the three aforementioned systems, Trane, Featuretools, and ATM and execution takes place in four stages. In the first stage, Trane is used to generate prediction problems, a list of training examples, testing examples and their labels. In the second stage, Featuretools is used to generate features for training and testing. In the third stage, we perform a quick analysis using different metrics and determine the most meaningful problems. In the fourth and final stage, we use ATM to select and tune a machine learning model for each problem of interest. In this chapter we will walk through each stage, show intermediate results and present our analysis. In doing so, we also demonstrate how a typical user would use MLF.

Data Background: We use the Yelp review dataset available on the data science competition website, Kaggle [10]. The yelp dataset contains 7 separate data tables. Information about the data tables is enumerated in 5.1 on page 50. The data spans from all the way back in mid-2004 to late-2017.

We focus our analysis on three of the tables in the data. We choose to exclude the files:

- business_hours.csv: We exclude business hours from our analysis for now, but
Table 5.1: Information about the 7 tables in the yelp dataset.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Brief Description of contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>yelp_business.csv</td>
<td>Contains information about each of the businesses.</td>
</tr>
<tr>
<td>yelp_review.csv</td>
<td>Contains information about all of the reviews on yelp and keys to join with the yelp_business and yelp_user table.</td>
</tr>
<tr>
<td>yelp_user.csv</td>
<td>Contains information about the users of yelp.</td>
</tr>
<tr>
<td>business_hours.csv</td>
<td>Contains information about the hours of operation for each business.</td>
</tr>
<tr>
<td>business_attributes.csv</td>
<td>Contains information about attributes of the business, for instance if a business has parking.</td>
</tr>
<tr>
<td>yelp_checkin.csv</td>
<td>Contains information about the checkins made at each business by users.</td>
</tr>
<tr>
<td>yelp_tip.csv</td>
<td>Contains information about tips that users have left for businesses. Namely, text data suggesting what to try or do at the business.</td>
</tr>
</tbody>
</table>

will use it later when generating features because the information contained in the data may be useful for feature generation.

- business_attributes.csv: We exclude the business_attributes.csv file because most of the data is null.

- yelp_checkin.csv: The yelp_checkin.csv file contains information about the users who checked in at a particular business, however there is no column containing precise information about the date the checkin occurred. We have no way to ascertain when entries in the file should be available. Therefore, we can not integrate this information into Trane properly. For example, if we were to include the data, we would not be able to segregate the data properly when calculating features and labels. If we can’t properly segment the cutoff time data, our system will, in essence, cheat by using the information that occurs later in time, to generate features.

- yelp_tip.csv: We exclude the file tip.csv because it only contains text information, which Trane does not work with, thus adding the data won’t provide additional value.
**Denormalization:** We denormalize the remaining three csv’s of interest, namely yelp_business.csv, yelp_review.csv and yelp_user.csv. The yelp_business.csv is indexed by `business_id`. Yelp_review.csv is indexed by `review_id`. Yelp_user.csv is indexed by `user_id`. Each row of the yelp business table contains a unique `business_id` and some attributes of the business, for example the average number of stars the business receives in reviews and the name of the business.

Each row of the yelp review table contains a unique review along with important information about the reviews. Some of the included fields are `stars`, the number of stars given in the review, `text`, the textual content of the review and `business_id`, the business the review is being left for. Each row of the yelp user table contains unique information about a user. Some of the attributes are `name`, the name of the user and `review_count`, the number of reviews that user has made on Yelp.

**Sampling:** There is a large amount of detailed information in the data. The reviews table contains over 5.2 million rows. The businesses table has over 170 thousand rows. The users table has over 1.3 million rows. We choose to avoid using this abundance of data and instead sample it, for two reasons. First, executing MLF on the almost 5 million rows of data would take a long time in our implementation. We did not implement Trane with performance in mind, so there are significant bottlenecks in it’s current execution. Second, the vast size of the dataset is not necessary to demonstrate MLF’s ability. MLF simply needs enough entities or training examples to be able to adequately train a classifier. How many data points are needed to adequately train a classifier? The number of data points needed heavily depends on the situation, but to provide a rough and general number, we would say 100. Thus, if we have over 100 entities we are fairly confident in MLF’s ability to generate classifiers with reasonable predictive power.

We focus our analysis solely on a sampled selection of the data from 200 businesses. We sample 200 business id’s from the yelp_business.csv file to use in our dataset. We then gather all the reviews and users linked to those businesses by using the `business_id`’s to sample the relevant reviews and the `user_id`’s in those relevant reviews to grab the relevant users. In total, we select 5185 reviews from 5059 users for
those 200 businesses. The sampled dataset provides 200 training samples. Therefore, we're confident that the sampled data will be enough for MLF to adequately show its ability.

Table 5.2: Selected fields of interest in the businesses table

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>business_id</td>
<td>Unique identifier for each business.</td>
</tr>
<tr>
<td>stars</td>
<td>The number of stars a business averages on its reviews.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the business.</td>
</tr>
<tr>
<td>categories</td>
<td>The categories the business caters to. Such as sports, hair or vehicles.</td>
</tr>
<tr>
<td>neighborhood</td>
<td>The neighborhood the business is in.</td>
</tr>
<tr>
<td>city</td>
<td>The city the business is in.</td>
</tr>
<tr>
<td>is_open</td>
<td>A boolean value indicating whether the business is still open</td>
</tr>
<tr>
<td>review_count</td>
<td>The number of reviews the business has received.</td>
</tr>
</tbody>
</table>

Table 5.3: Selected fields of interest in the users table

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_id</td>
<td>Unique identifier for each user.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the user.</td>
</tr>
<tr>
<td>review_count</td>
<td>The number of reviews a user has left.</td>
</tr>
<tr>
<td>yelping_since</td>
<td>The date a user started their account.</td>
</tr>
<tr>
<td>compliment_funny</td>
<td>The number of compliments the user has received denoting their reviews as funny.</td>
</tr>
</tbody>
</table>

Data Cleaning: The data is well formatted and clean. Each of the fields contains no null nor missing values. That makes our job a lot easier, because we don’t have to find and replace values or delete certain examples due to a lack of data.

The denormalized table has 42 columns and there are some overlappings of column names. For instance, the reviews table has a field called stars and the businesses table also has a field called stars. The two columns both represent different values. The stars field for a business represents their average stars rating, whereas in the
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>review_id</td>
<td>Unique identifier for each review.</td>
</tr>
<tr>
<td>date</td>
<td>The date a review was made.</td>
</tr>
<tr>
<td>text</td>
<td>The text of the review</td>
</tr>
<tr>
<td>stars</td>
<td>The number of stars a review gives.</td>
</tr>
<tr>
<td>business_id</td>
<td>The unique identifier for the business a review was left at.</td>
</tr>
<tr>
<td>user_id</td>
<td>The unique identifier for the user who left a review.</td>
</tr>
<tr>
<td>useful</td>
<td>The number of votes each review has received denoting the review as useful.</td>
</tr>
<tr>
<td>funny</td>
<td>The number of votes each review has received denoting the review as funny.</td>
</tr>
</tbody>
</table>

reviews table, stars is the number of stars a review gives. Despite the name overlap, the data is successfully merged and the columns are automatically renamed to prevent a collision or overwrite of data. One column is renamed to stars_x and the other to stars_y, with neither keeping the original name. We must be careful to inspect the columns in the merged data if we wish to use a column that has been automatically renamed as one of the label generating columns. If we wish to use the average stars a business received, we must determine the new column name and set it to the original name, stars or, to set the label generating column to the new automatically determined name. We perform this step for the stars in the review table, as we wish to use that column as the label generating column. With that simple column overlap solved and the irrelevant files ignored, the data is clean and ready to be used.

**Data Exploration:** We will be interested in predicting characteristics related to the stars column later in the walk through, therefore it would be valuable to understand some of the data. Here we see the distribution of stars typically given to restaurants. We would expect a uniform distribution, but it’s clear that people prefer to leave either very positive or very negative reviews, hence the significantly higher number of 5 star and 1 star reviews. As well, we see that people are generally quite biased towards leaving very positive reviews. The number of 4 star and 5 star reviews significantly
Now that we have assembled our dataset and understand its fields, we are ready to begin the first step of MLF. The first step is Trane and the system requires four inputs.

5.1 Trane

Inputs to Trane: The four inputs to Trane are datasets, relationships, four column choices and meta information.

Datasets: The first input is simply the filenames containing the data we wish to use. Dataset: [yelp_business.csv, yelp_user.csv, yelp_review.csv]

Relationships: Each business has many reviews made about it, therefore there exists a one to many relationship between those two tables. Similarly, each user makes many reviews, so there exists a one to many relationship between those two tables as well. Note, that we do not need to specify the type of relationship, we must simply provide the filenames and join keys that should be used for the tables. For instance, for the business and review table we specify the join key for both tables as the business_id. Note, the information in the first input datasets, is also contained in the relationships input. We choose to still specify the datasets input, because in case of a single data table input, there will be no relationships. We could add a special way to indicate the input is a single table in the relationships input, but we believe
the current setup of inputs is easily understandable and requires hardly any additional effort.

Relationships:

```
[(yelp_business.csv, business_id, yelp_review.csv, review_id),
 (yelp_user.csv, user_id, yelp_review.csv, user_id)]
```

**Meta Information:** The third input is a file containing meta information. It is simple to infer the types of data using the descriptive column names provided in the dataset. For instance, the field `text` which contains a user's review is clearly a text value, `user_id` is an identifier and `stars` is a number, specifically a float. We feed the file into a utility function defined by Trane, `file_to_table_meta`, in order to transform the file into a table meta object which Trane will use.

The meta.json file we created for these three tables is available in the appendix on page 77.

Then, we convert the meta file to a table meta object which Trane expects.

```
table_meta = file_to_table_meta("yelp\_meta.json")
```

**Meta information:** `table_meta`.

**Four Columns:**

**Entity id column** We could use the `user_id` as our entity id column. This would allow us to generate prediction problems about user behavior. For instance, the question: predict the next `business_id` a user will leave a review for. This could be a very interesting problem for business owners to solve. Answering this question would help businesses better understand their target customers. The question may also help user's identify other businesses a user may be interested in. For example, consider a user that commonly eats at Chinese restaurants. The features selected to predict the restaurant a user will eat at next will include characteristics of the restaurants the user frequents, namely Chinese restaurants. Thus, during prediction, the classifier will predict restaurants with features similar to the Chinese restaurants he enjoys and possibly predict new restaurants for the user to try out. We could also select the entity id column to be the `business_id` column. This would allow us to predict characteristics of businesses. For example, trying to predict how many reviews
a business will receive in the next few days. We decide to set the entity id column to be `business_id`.

**Label generating column** Given, we have selected the entity id column to be `business_id`, we should focus on selecting a column that we want to predict characteristics of, specifically related to businesses. For instance, we may want to predict the number of stars in the next review a business receives. Thus, we decide to set the label generating column to be `stars`. Specifically, the `stars` column in the reviews table, not the business table.

**Filter column** We may only want to consider users who have recently joined Yelp. The business may want to try to predict the behavior of new users, because they assume more experienced and frequent users are less likely to be influenced into trying a new business, having become set in their ways. Thus, we could select the filter column to be the `yelping_since` column in the users table. We may also want to filter by the number of reviews a user has left. A user who leaves many reviews is likely experienced with Yelp and writes thoughtful and relevant reviews. Perhaps a business owner wants to ignore a specific user’s review and would set the filter column to be `user_id`. This way a business could ignore the reviews of a specific user they know to be malicious or with malintent, for instance an account created by the businesses competitor. We decide to set the filter column to be the `user_id` in the users table.

**Time column** There is only one reasonable choice of time column: `date` in the review table. The review table has a many to one relationship with each of the other two tables. Therefore, it contains unique information for every entry of the merged table. As well, there are no other time columns we may use that would make sense. For instance, using `yelping_since` as the time column would segment data to be available based on the time a user first joined Yelp. Segmenting the data by the time the user joined Yelp is not meaningful or useful ensuring information leakage does not occur. We decide to set the time column to `date`.

1. **Entity id column:** `business_id`
2. **Label generating column:** `stars`
3. Filter column: user_id
4. Time column: date

Now that we have specified all the inputs required for Trane to generate problems and labels, we can execute the system and examine the output.

5.1.1 Outputs from Trane

Trane generated 66 valid prediction problems or questions.

How are problems generated?: Trane first enumerated every possible question it could ask by performing a cartesian product over the 4 classes of operations. The cartesian product of the classes of operations will result in $5 \times 6 \times 3 \times 5 = 450$ prediction problems. Trane prunes operations that are not possible according to the type information we know from the meta information input. For instance, because user_id is the filter column and it’s of type identifier, we may only use operations in that step, which are able to accept inputs of type identifier. The label generating column is stars which we know is a float. Therefore, we may only use row operations that can accept a floating point number as input. Through this pruning process, the 450 possible prediction problems are reduced to 66 valid prediction problems. These 66 prediction problems are valid because they all may be executed on the data to produce a label. Trane also generates hyper parameters for each of the problems.

5.1.2 Discussion

What do the problems look like?: The 66 prediction problems cover a wide variety of applications. We sifted through the 66 problems by hand to select problems generated by Trane for the data and placed them in three categories: Meaningful, Impractical and In-between. Below we will describe what we mean by these categories, and provide examples within each category. For each example, we provide three pieces of information. First, the series of operations that form the problem’s structure. Second, a translation of the problem into English generated by hand by
us. Third, the translation that Trane automatically generates for each problem. The translation algorithm Trane uses simply iterates through the four operations that make up a prediction problem by integrating the semantic use of each operation into a semiunderstandable sentence.

Note, the time period referred to for the questions is the length of time in the testing period. In this case, the cutoff times have been dynamically and automatically determined by Trane to be where the last 20% of the data is available for a given entity. Thus, the time period varies for each business.

Wellphrased, meaningful and of practical interest: Some of the questions are well-phrased, meaningful and of practical interest. Now, we show some example problems generated by Trane, that are Meaningful.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>Predict the last minus first of the differences of the exponentiation of stars to the power of 1 in reviews for a business.</td>
</tr>
<tr>
<td>Trane’s translation</td>
<td>For each business_id, predict the last minus first the fluctuation of the exp of stars, after 80% of the entity’s data has elapsed.</td>
</tr>
</tbody>
</table>

Because the hyper parameter of the exponential row operation is 1, this problem is simply asking to find the difference between two differences. The first difference is between the next two reviews stars rating. The second difference is between the last two reviews stars rating. The difference of the difference is akin to approximating a second derivative for the number of stars received in reviews over time. If the difference of the difference is positive, the rate at which stars is growing will increase with time. If the difference of the difference is negative, the rate at which the stars a business receives in reviews is growing will decrease with time.

<table>
<thead>
<tr>
<th>Problem</th>
<th>AllFilterOp ⇒ IdentityRowOp ⇒ IdentityTransformationOp ⇒ FirstAggregationOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>Predict how many stars the next review will have.</td>
</tr>
<tr>
<td>Trane’s translation</td>
<td>For each business_id, predict the first stars, in the next time period.</td>
</tr>
</tbody>
</table>
A business would be interested in predicting how many stars their next review will have. Businesses can compare this predicted value to their current average stars rating. If the predicted number of stars is above their average rating, a business can learn that their customers are likely to rate the store higher in the future, implying that the business owners are doing a good job. On the contrary, if the predicted number of stars is below their average rating, they will know their business needs some work. In either case, the business owner is provided with valuable information.

<table>
<thead>
<tr>
<th>Problem</th>
<th>AllFilterOp ⇒ LessRowOp ⇒ IdentityTransformationOp ⇒ FirstAggregationOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>Predict if the next review will be less than 5 stars.</td>
</tr>
<tr>
<td>Trane’s translation</td>
<td>For each business_id, predict whether the op stars is less than 5, after 80% of the entity’s data has elapsed.</td>
</tr>
</tbody>
</table>

This problem is similar to the prior one, but the output will have fewer classes, due to the row operation. Namely, there are only two possible values: true and false. Notice, the problem sets a very high bar. Only if a review has 5 stars, in other words a perfect score will the answer be true. However, this is not too surprising, given the distribution of stars we saw in the data exploration. We know a lot of the stars ratings given in reviews are 5. If a business has recently made some changes, they may expect their customers to like the changes. A shift in the average number of stars received metric would be difficult to perceive. The average number of stars a business has is bogged down by a lot of reviews that occurred in the past. Therefore, it will take a lot of new reviews with different ratings to change the metric significantly. Businesses can use the information provided by the answer to this problem to know immediately how they are expected to perform in the short-run.

<table>
<thead>
<tr>
<th>Problem</th>
<th>AllFilterOp ⇒ IdentityRowOp ⇒ IdentityTransformationOp ⇒ CountAggregationOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>Predict how many reviews a business will receive in the next time period.</td>
</tr>
<tr>
<td>Trane’s translation</td>
<td>For each business_id, predict the number of records, after 80% of the entity’s data has elapsed.</td>
</tr>
</tbody>
</table>
A business would certainly like to know how many reviews they will receive in a future period of time. From this information, they can simply determine the frequency at which they will receive reviews by dividing the answer by the length of the period. For instance, if the answer is 24 reviews over a period of 4 days, they can expect an average of 6 reviews a day. The number of customers is likely positively correlated with the number of reviews. A business with many customers will likely have many more reviews than a business with very few customers. Therefore, a business could use the frequency of reviews they receive as a rough forecast for the number of customers they will receive. If a business knows tomorrow will be a very busy day with lots of customers, they can try to call in extra staff or load the shelves with more goods.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>Predict the next difference in the exponentiation of the stars in a review to the power of 1.</td>
</tr>
<tr>
<td>Trane’s translation</td>
<td>For each business_id, predict the first the fluctuation of the exp of stars, after 80% of the entity’s data has elapsed.</td>
</tr>
</tbody>
</table>

This problem sounds more convoluted than it truly is. The exponentiation operation is to the power of 1. Anything to the power of 1 is unchanged, so we may simply ignore it. The problem predicts the difference in stars between the next two reviews. If this difference is positive, a business knows they are likely to improve in the future. However, if the difference is negative, the business knows they should put some work in to their product or company to turn the trend around. Note, this problem will likely end up with a lot of 0 values for its labels, because there is not much granularity in the review values. For instance, with only 5 possible choices for rating and a lot of the reviews set to 5 stars, the predicted future difference is likely to be small or 0. However, if the stars data was on a scale from 1 to 100, the labels would likely be more varied and valuable.
The difference between the last and first review's number of stars can provide an idea of how the business is trending over a long period of time. This problem is akin to the one we saw earlier, where we predict the difference in the next two reviews, but this problem is over a much larger time span. Given the larger time span, we expect more valuable answers, because it's unlikely a business significantly changes from one review to the next, however over a period of a few weeks, the amount of stars they receive in reviews may be trending in a certain, noticeable direction.

*Impractical:* Other questions are not meaningful and have little practical interest.

<table>
<thead>
<tr>
<th>Problem</th>
<th>AllFilterOp ⇒ IdentityRowOp ⇒ IdentityTransformationOp ⇒ LMFAggregationOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>Predict the difference in the amount of stars between the last and first review in a time period.</td>
</tr>
<tr>
<td>Trane's translation</td>
<td>For each business_id, predict the last minus first stars, after 80% of the entity's data has elapsed.</td>
</tr>
</tbody>
</table>

Predicting the total number of stars a business will receive is not particularly helpful. A business may receive 500 total stars from 500 reviews, each review averaging a single star, the lowest rating. Another business may receive 10 total stars from two reviews, each review averaging 5 stars, the highest rating. The sum of stars does not provide us with any particularly concretely useful information.
Table 5.5: The number of each kind of problem identified.

<table>
<thead>
<tr>
<th></th>
<th>Meaningful</th>
<th>Impractical</th>
<th>In-between</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>42</td>
<td>14</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>Predict the number of unique stars ratings present for a business after exponentiating each stars review count to a threshold power.</td>
</tr>
<tr>
<td>Trane’s translation</td>
<td>For each business.id, predict the number of records, after date 2015-01-01.</td>
</tr>
</tbody>
</table>

This problem simply asks to predict the number of unique stars ratings a business observes. This could be perceived as a kind of proxy for the standard deviation of reviews, but is not truly meaningful or of great use to a business, because unlike standard deviation, it will not take into account the number of instances seen in each class. For instance, if a business receives five one star reviews and ten five star reviews. There is a lot of variance in the reviews observed, but this question would generate a label of two. Now consider, a business that receives one hundred 3 star reviews, one five star review and two one star reviews. The question would generate a label of 3, which we may interpret as meaning the data is more varied, however it is clearly not more varied than the first example business’s data.

**In-between:** Finally, there are some problems that are inbetween being meaningful and being impractical.

<table>
<thead>
<tr>
<th>Problem</th>
<th>AllFilterOp ⇒ NeqRowOp ⇒ IdentityTransformationOp ⇒ LastAggregationOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>Predict if the last review will not have stars equal to 5.</td>
</tr>
<tr>
<td>Trane’s translation</td>
<td>For each business.id, predict the op stars is not equal to 5, after 80% of the entity’s data has elapsed.</td>
</tr>
</tbody>
</table>

This problem is trying to predict whether the last review’s stars received in a time period will not be perfect. This problem may have some value, but it’s not clear how a business could take advantage of this information.
Can these problems be executed?: Trane generates answers to the prediction problems, commonly referred to as labels. The labeller generates labels by segmenting each business_ids data into separate data tables. Each business_id's table will contain many entries. Each entry represents a review that was made. Trane first separates the data using the testing cutoff time then executes each of the problems on every entity's data. For example, let's execute business_id 2FA50X3's data to generate a training label. First the data is segregated by cutoff time. Because we are generating the training label, we use only the data available before the testing cutoff time. Then the data is sent through the filter operation which transforms the entity's data based on the filter criterion and passes. The filtered dataframe is now sent into the row operation instance for execution. The process continues for the remaining transformation and aggregation operation, resulting in a single value, the label. Trane generates labels for every business_id. Thus, we generate 200 training labels and 200 testing labels for each problem. Trane generates all these labels for every prediction problem. We generate 66 tables of labels, one for each problem. Each table contains every business_id mapped to its corresponding training label and testing label as well as the training and testing cutoff time.

Identical problem labels Some of the problems generated by Trane produce identical labels despite having unique underlying representations. Note, the underlying representations are simply the order of operations. For instance, consider two problems. They are identical besides their row operation. One problem has a greater row operation, the other problem has an equals row operation. The last operation, the aggregation operation, is a count, which simply counts the number of rows in the data. Despite the different row operations, the counts for the two problems will remain the same, thus they generate identical labels. It is possible that two questions ask something different, yet have identical labels, for instance, is x greater than 2 and is x greater than 3 will have the same value if the only x's input are above 3. However, given the strong similarity between questions with identical labels for every sample, we deem it prudent to prune them. We identify problems and labels of this nature and prune them, so they are only used once. 33 questions remain after the removal of
Table 5.6: The number of each kind of problem identified.

<table>
<thead>
<tr>
<th>Meaningful</th>
<th>Impractical</th>
<th>In-between</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>Identical and Meaningful</td>
<td>26</td>
<td>5</td>
</tr>
<tr>
<td>Unique and Meaningful</td>
<td>16</td>
<td>9</td>
</tr>
</tbody>
</table>

all those which produce identical labels or labels with a single class. Note, we may use the continuous labels by simply adding a row operation to the problem in order to transform the regression to a classification. For instance, instead of predicting the difference in the number of stars in the next two reviews, we would predict if the difference in the number of stars in the next two reviews is less than 3. 48% of those unpruned questions are meaningful. Furthermore, 33 questions is not a tremendous amount of questions for a human to sift through, as opposed to the 450 questions originally generated. In other words, a human user may just choose to perform the later step of determining meaningful problems by hand, when applying MLF.

**Two problems which produce identical labels.**

| Problem | AllFilterOp ⇒ GreaterRowOp ⇒ IdentityTransformationOp ⇒ CountAggregationOp |
| Problem | AllFilterOp ⇒ EqRowOp ⇒ IdentityTransformationOp ⇒ CountAggregationOp |

The labeller generates training labels and testing labels as well as determines the dynamic cutoff times for each entity. Here are some of the outputs from the labeller for a variety of problems.

The cutoff times are being automatically and dynamically chosen for every entity by the cutoff time generation algorithm. The algorithm intelligently selects the cutoffs for each entity based on the amount and time span of the data available in each entity. For example, consider a business that was started in 2009 and closed in 2013. Trane will select cutoff times for that business to be around 2011 and 2012. Another business started in 2013 which is still open today will have cutoff times chosen by Trane around
Table 5.7: Labels for the problem: predict how many stars the next review will have. Note how the cutoff times change for each entity.

<table>
<thead>
<tr>
<th>business_id</th>
<th>training_labels</th>
<th>test_labels</th>
<th>training_cutoff_time</th>
<th>test_cutoff_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>-McKyjNSqS1h9dDJH3dyUA</td>
<td>5</td>
<td>4.0</td>
<td>2009-02-15</td>
<td>2013-02-23</td>
</tr>
<tr>
<td>-nHkhliuerqmfBG3v2v9O-g</td>
<td>4</td>
<td>1.0</td>
<td>2016-11-06</td>
<td>2017-03-24</td>
</tr>
<tr>
<td>-ooEO2YqDQVYNHnSF2BPfw</td>
<td>3</td>
<td>5.0</td>
<td>2013-11-18</td>
<td>2016-01-23</td>
</tr>
<tr>
<td>0FMKDOUBTJT1x87OKYGDTg</td>
<td>4</td>
<td>5.0</td>
<td>2015-07-03</td>
<td>2016-12-01</td>
</tr>
</tbody>
</table>

2016 and 2017. Currently, the algorithm will automatically use 60% of the data for training features, 20% of the data for testing features and training labels, and the final 20% of the data for testing labels. There are 200 total entities and thus 200 labels for each problem. Some entities, when restricted by the cutoff times, no longer have enough data to generate a label and give a null value. We drop these null values from our labels. We noticed, 35% of the labels were being removed, because of this issue with cutoff times. Thus, we set the dynamic cutoff times to leave 40% of the data for the first phase, 30% for the second phase and 30% for the final phase. Now, only 17.5% of the labels are removed. With labels and problems, we move on to the next stage of MLF, generating features using FT.

5.2 Featuretools

Inputs to Featuretools: Featuretools requires four inputs for execution. Entities, relationships, cutoff times and a target entity. Many of the inputs we give to Featuretools are similar to the inputs provided to Trane.

Entities: Entities are simply the dataframes we load from the csv files, which were specified as the datasets input to Trane. We use two of the additional data files we initially ignored, tips.csv and business_hours.csv. The two files likely contain useful information for characterizing businesses. The five dataframes now, are user, business, tips, business hours and review.

Relationships Relationships, denotes the same information we passed in to Trane, in a slightly stricter format. The first entry in the tuple must be the parent entity.
and the second entity must be the child entity. For instance, when specifying the relationship between the business dataframe and the review dataframe, the business dataframe must come first because it is the parent entity.

**Target entity:** The target entity is the entity we are interesting in predicting characteristics of, in this case business. Note, when we define an entity, we must define the index column. The index column contains a unique identifier for each row of the data. For example, for the user table we specify the index column as user_id. Additionally, we may specify the time column. The time column of an entity specifies when the data in that row became available. For instance, the time column for the review entity is date, because that is when information contained in the review became available. For the users entity, the time column is yelping_since, because that field defines when that user’s information became available. Knowing when data is available will come in handy for preventing information leakage and ensuring cutoff times are enforced.

**Cutoff times:** Cutoff times are specified to FT to prevent information leakage. The same cutoff times used in Trane are passed to FT. FT ensures only data available before the cutoff time is used to generate the features. For instance, data past Jan 20 2014 won’t be available to FT to generate training features, because the data after Jan 20 2014 is supposed to be set aside and used for generating training labels. If FT used the data past Jan 20 2014 to generate the training features, it would be too easy for FT to generate very accurate features, because it can see the data from which labels will be made. Thus, the feature engineering system would, in essence, be cheating. Deep Feature Synthesis is an algorithm defined in FT that performs the feature generation. We execute the DFS algorithm twice. Once with the training cutoff times specified for each business_id. Once with the testing cutoff times specified for each business_id.

DFS is designed to generate features using multiple tables. DFS follows the relationships across tables to generate three types of features, entity features, direct features and relational features. Feature tools recursively operates to generate all three kinds of feature for the entities present.
5.2.1 Discussion

How does feature generation work?: Let's consider generating features with DFS using only three entities: users, businesses and reviews. Imagine three new tables, one for each entity. Each table is initially empty but will be used to contain the features for its corresponding entity. Consider generating features for the business entity. The algorithm first follows the relationships from the business entity and arrives at the reviews entity. It then, tries to generate features for the reviews entity. In order to generate features, it must first follow all relationships from the review entity. There are two relations for the reviews entity. Both relationships are forward and point to the users and business entity. Because we are already exploring the business entity, the algorithm follows the relationship to the user entity. The user entity contains no further relationships, so we have reached the base of recursion. FT now generates entity features for the table. Then, these features are directly carried over to the features for the review entity. Now, features are calculated for the review entity using both the direct features from the user entity and entity features constructed from data present in the review entity. Finally, the features are transformed across the original relationship to the business table. Because the relationship is many to one, the features are transformed by functions to aggregate the features in the review table that correspond to a single entry in the business table. Note, we may specify the types of functions FT will use to aggregate features when features are transformed from a many entity to a one entity as they are done between the reviews entity and the businesses entity. We choose to ignore some of the aggregation functions, such as Skew and percent true, because we believe those features will not serve a useful purpose in predicting information about businesses. We also want to minimize the number of features in order to prevent overfitting to the training data, thereby reducing the classifier's ability to generalize. We decide to keep other aggregation functions like count and max, because those will serve well to represent the data from the reviews, for instance counting the number of users who made a review at a particular business would be a useful feature. [4]
Now we show some sample code that would execute DFS as described above.

```python
import featuretools as ft

es = ft.EntitySet(id = "all-entities")
es.entity_from_dataframe(entity_id = "business",
    dataframe = sampled_yelp_business_df,
    index = "business_id")
es.entity_from_dataframe(entity_id = "reviews",
    dataframe = sampled_yelp_review_df,
    index = "review_id",
    time_index = "date")
es.entity_from_dataframe(entity_id = "users",
    dataframe = sampled_yelp_user_df,
    index = "user_id",
    time_index = "yelping_since")

# Note: (parent entity, child entity)
re1 = ft.Relationship(es['business']['business_id'], es['reviews']['business_id'])
re2 = ft.Relationship(es['users']['user_id'], es['reviews']['user_id'])
es.add_relationship(re1)
es.add_relationship(re2)
```

Then, we specify the relationship between the three tables, as follows. Keep in mind the parent entity comes first, followed by the child entity. The first relationship's parent entity is business because each business has many reviews. The second relationship's parent entity is users because each user makes many reviews.

```python
# Note: (parent entity, child entity)
re1 = ft.Relationship(es['business']['business_id'], es['reviews']['business_id'])
re2 = ft.Relationship(es['users']['user_id'], es['reviews']['user_id'])
es.add_relationship(re1)
es.add_relationship(re2)
```

We obtain the training and testing cutoff times from the output Trane provided us.

```python
sampled_entity_ids = merged_df[entity_id_column].unique()
training_features_cutoff_times = labels[0][['training_cutoff_time', 'business_id']]
test_features_cutoff_times = labels[0][['label_cutoff_time', 'business_id']]
```

Then, we execute the DFS algorithm twice, once with the training cutoff times and again with the testing cutoff times. Note, we also transform the returned features into one-hot features, so they may be used for training a logistic regression model.

```python
training_features_matrix, training_feature_definitions =
    ft.dfs(entityset = es,
        target_entity = "business",
        cutoff_time = training_features_cutoff_times)
training_features_matrix, training_features =
    ft.encode_features(
        training_features_matrix,
        training_feature_definitions)

test_features_matrix, test_feature_definitions =
    ft.dfs(entityset = es,
        target_entity = "business",
        cutoff_time = test_features_cutoff_times)
test_features_matrix, test_features =
```
5.2.2 Outputs from FT

FT generates training and test features. The recursive nature of feature generation along with the 5 entities we input along with the multitude of relationships specified allow FT to generate a whopping 188 features for each of the 200 businesses.

MLF has now executed the first two stages. We have training labels, training features, testing labels and testing features. We move on to ROC curve analysis in order to automatically identify the most meaningful problems. Below, we provide examples features that were calculated for the businesses entity, originating from the entities business, users and reviews. Note, the column names, which provide details about how the features were made.

Table 5.8: A handful of the features generated by DFS from the business entity.

<table>
<thead>
<tr>
<th>business_id</th>
<th>stars</th>
<th>review_count</th>
<th>is_open</th>
<th>categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJFdWX908N8Yc2XG0Lky8A</td>
<td>4.0</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nigYwB_m1TQ1WosjSWi-Hw</td>
<td>3.0</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2pD9wZWXDNsZfMXd8rQtg</td>
<td>2.5</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>zV_aclADljx2KOql9F_FTw</td>
<td>3.0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.9: A handful of the features generated by DFS from the users entity.

<table>
<thead>
<tr>
<th>business_id</th>
<th>MAX(reviews.users .average_stars)</th>
<th>MAX(reviews.users .compliment_hot)</th>
<th>MAX(raviews.users .compliment_more)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJFdWX908N8Yc2XG0Lky8A</td>
<td>4.21</td>
<td>154</td>
<td>16</td>
</tr>
<tr>
<td>nigYwB_m1TQ1WosjSWi-Hw</td>
<td>3.98</td>
<td>4551</td>
<td>259</td>
</tr>
<tr>
<td>2pD9wZWXDNsZfMXd8rQtg</td>
<td>4.26</td>
<td>495</td>
<td>39</td>
</tr>
</tbody>
</table>

5.2.3 ROC Curve Generation

We attempt to train a Logistic Regression classifier for every problem. Of the 66 problems, 8 problems initially fail to train because the labels contain continuous
Table 5.10: A handful of the features generated by DFS from the reviews entity.

<table>
<thead>
<tr>
<th>business_id</th>
<th>MIN</th>
<th>MEAN</th>
<th>MEAN</th>
<th>MEAN</th>
<th>MEAN</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(reviews .cool)</td>
<td>(reviews .stars)</td>
<td>(reviews .useful)</td>
<td>(reviews .funny)</td>
<td>(reviews .cool)</td>
<td>(reviews)</td>
<td></td>
</tr>
<tr>
<td>EJFdrX908N8Yc2XG0Lky8A</td>
<td>0</td>
<td>5.000000</td>
<td>2.000000</td>
<td>0.000000</td>
<td>0.500000</td>
<td>2</td>
</tr>
<tr>
<td>nigYwB..m1TQ1WosjSWi-Hw</td>
<td>4</td>
<td>5.000000</td>
<td>5.000000</td>
<td>1.500000</td>
<td>4.000000</td>
<td>2</td>
</tr>
<tr>
<td>2pD9wZWXDNslMXd8rQtg</td>
<td>0</td>
<td>3.666667</td>
<td>3.666667</td>
<td>2.000000</td>
<td>0.000000</td>
<td>3</td>
</tr>
<tr>
<td>zV_aclADLjx2KQql9F_FTw</td>
<td>0</td>
<td>2.666667</td>
<td>1.333333</td>
<td>0.333333</td>
<td>0.666667</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.11: The five problems identified by Trane as the most meaningful.

<table>
<thead>
<tr>
<th>Meaningful-ness rank</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Predict the number of reviews a business will receive in the following time period.</td>
</tr>
<tr>
<td>2</td>
<td>Predict if there exist both reviews with stars on either side of the threshold 5, or only one side.</td>
</tr>
<tr>
<td>3</td>
<td>Predict the difference in the number of reviews seen with the highest stars amount seen for this business and the number of reviews seen with the lowest stars amount for this business.</td>
</tr>
<tr>
<td>4</td>
<td>Predict the total number of stars a business will receive across all reviews received in a future time period.</td>
</tr>
<tr>
<td>5</td>
<td>Predict the difference of stars between the last and second last review.</td>
</tr>
</tbody>
</table>

Values. For instance, the problem: predict the next trip’s fare, will have a wide range of float values as labels. To handle this issue, we transform the problem to a classification by appending a row operation to the end. For instance, we append a GreaterRowOperation to threshold the output and transform floats into one of the two boolean values, True and False.

Note, the problems with meaningfulness ranks of 1, 3, 4 and 5 are being compared to the 60% quantile value because we chose to modify the labels produced by Trane to generate binary classifications. Thus, we simply appended a Greater Row Operation with the hyper parameter set to the specific quantile value. Users may alter prediction problems by using the intermediary prediction problems JSON file generated by Trane.

We will select the models with an AUC score nearest 0.5, because that indicates the problems are very difficult to solve. Below, we list the five problems with the
lowest AUC scores. In other words, the problems MLF determines to be the most meaningful. At this time, the user may also browse through all the problems and ROC curves generated to determine which problems are most interesting for them and ignore Trane’s suggestions.

<table>
<thead>
<tr>
<th>Problem</th>
<th>AllFilterOp ⇒ IdentityRowOp ⇒ IdentityTransformationOp ⇒ CountAggregationOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>Predict the number of reviews a business will receive in the following time period.</td>
</tr>
<tr>
<td>Trane’s translation</td>
<td>For each business_id, predict the number of records, after date 2015-01-01.</td>
</tr>
</tbody>
</table>

The problem that MLF determines to be the most meaningful, was also determined to be meaningful via our a priori analysis of each generated problem. The Area under the curve (AUC) score is 0.5. 0.5 is the lowest AUC score a classifier can obtain. This implies the problem is very difficult to predict because the classes of output are nearly impossible to separate.

<table>
<thead>
<tr>
<th>Problem</th>
<th>AllFilterOp ⇒ GreaterRowOp ⇒ ObjectFrequencyTransformationOp ⇒ CountAggregationOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>Predict if there exist both reviews with stars on either side of the threshold 5, or only one side.</td>
</tr>
<tr>
<td>Trane’s translation</td>
<td>For each business_id, predict the number of records, after date 2015-01-01.</td>
</tr>
</tbody>
</table>

MLF determines this problem to be the second most meaningful. The problem has an AUC of 0.51. We determined it to be in-between via our a priori analysis. Note, the problem is a kind of proxy for how varied the dataset is. However, the problem only predicts if there are reviews that fall on either side of the threshold of 5 stars. It’s clearly a difficult problem to predict, but not very useful.
MLF determined this problem to be the third most Meaningful. The problem has an AUC of 0.53. We determined it to be meaningful via our a priori analysis. The problem is meaningful because business owners can get a sense for how many more or less users really like a business versus how many users really dislike a business.

MLF determined this problem to be fourth most meaningful. The problem has an AUC of 0.54. We determined it to be In-between via our a priori analysis. Knowing the total number of stars is partially useful to a business owner, but without knowing how many reviews contributed towards that many stars, the answer to the question isn’t particularly useful.

MLF determined this problem to be fifth most meaningful. The problem has an AUC of 0.55. We determined it to be Meaningful via our a priori analysis. We believe its meaningful, because it gives the business owner a sense of the direction the number
of stars in a review will trend into the future.

The five most Meaningful problems found by MLF contained three truly Meaningful problems and two In-between problems. Based on the number of each type of problem identified in table 5.11, we would expect to see roughly 2 Meaningful problems, one In-between and one Impractical. While, we certainly can’t draw statistics from this small sample size, the evidence is promising.

5.3 ATM

We can now pass the 5 identified problems, training features and labels to ATM to generate and tune a model. ATM performs an intelligent cartesian search over a variety of possible modeling methods, such as an SVM or Random Forest model. Each model has hyper parameters. The hyper parameters can be variables such as number of layers to use or number of epochs. Each hyperparameter selection may have conditional hyper parameters. That is, parameters which must be specified for certain choices of hyper parameter. The choices for model, hyperparameter and conditional hyper parameter form a model pipeline capable of performing predictions. ATM enumerates over these possible pipelines to generate models. The models are evaluated on the training data. Of all the models generated, we select the model with the highest F1-score on the training data.

We ran ATM using the training and test data we have built with Trane and FT. We specified a budget of 1000 to ATM, which means ATM enumerated and tried 1000 models and variants of them. It took roughly 7 minutes for ATM to generate the optimal model.

Finally, we take each optimal model and perform predictions on the test data. We see improvement in the F1 score for classification of all problems, but one. The problem with a lower F1 score has a higher accuracy.

The problems are all unique and most of them are meaningful and interesting. These are promising results for MLF. We can now pass the features and labels for our problems of interest to ATM to generate optimized predictive models.
Table 5.12: F1 and Accuracy score comparisons for basic Logistic regression models and ATM’s best generated model.

<table>
<thead>
<tr>
<th>Problem</th>
<th>F1 Score with LogReg</th>
<th>F1 score with ATM</th>
<th>Accuracy with LogReg</th>
<th>Accuracy with ATM</th>
<th>Number of Positive Labels</th>
<th>Number of negative Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0625</td>
<td>0.166</td>
<td>0.818</td>
<td>0.794</td>
<td>32</td>
<td>133</td>
</tr>
<tr>
<td>2</td>
<td>0.305</td>
<td>0.330</td>
<td>0.503</td>
<td>0.479</td>
<td>123</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>0.111</td>
<td>0.111</td>
<td>0.806</td>
<td>0.903</td>
<td>33</td>
<td>132</td>
</tr>
<tr>
<td>4</td>
<td>0.235</td>
<td>0.222</td>
<td>0.842</td>
<td>0.921</td>
<td>33</td>
<td>132</td>
</tr>
<tr>
<td>5</td>
<td>0.071</td>
<td>0.190</td>
<td>0.800</td>
<td>0.885</td>
<td>22</td>
<td>99</td>
</tr>
</tbody>
</table>

5.4 Conclusion

We have built a simple to understand, reliable and modular implementation of Trane. The implementation is no longer a black-box and instead is designed to be used and added on to via collaboration with the community. We hope collaborators will create new operation implementations that will improve the meaningfulness of the problems Trane generates. Future work may also optimize the computation of Trane in order to increase the size of the datasets it may handle, which in turn will allow for more training and testing data and likely, higher accuracies for classifiers. We have demonstrated some of the potential of Trane in the context of a larger MLF system. The problems and classifiers MLF generated show the kind of role MLF may play for data scientists and laypeople.
Chapter 6

Contributions

6.1 Expected Contributions

In this thesis, we have made four key contributions.

1. A ground-up redesign and implementation of Trane, built with additional features and aspects of good programming practice such as modularity and clarity.
3. A significant reduction in the number of meaningless problems generated.
4. End-to-end demonstration of MLF’s viability.
5. The release of public demos so that users can try MLF on their own data.

6.2 Conclusion

Trane is a system that takes a dataset and relevant meta information as input to create two important outputs: a list of relevant prediction problems and the labels required to train a machine learning algorithm. It’s a general system that can be used on time varying datasets to obtain insights into the data. Trane has been combined with FT and ATM to create MLF, which has the potential to greatly improve data scientists’ productivity and the accessibility of machine learning.
Appendix A

Tables

```json
{
    "tables": [
        {
            "fields": [
                {
                    "name": "review_id", "type": "text"
                },
                {
                    "name": "user_id", "type": "text"
                },
                {
                    "name": "business_id", "type": "text"
                },
                {
                    "name": "stars", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "date", "type": "datetime"
                },
                {
                    "name": "text", "type": "text"
                },
                {
                    "name": "useful_x", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "funny_x", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "cool_x", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "text", "type": "text"
                },
                {
                    "name": "review_count_x", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "yelping_since", "type": "datetime"
                },
                {
                    "name": "friends", "type": "text"
                },
                {
                    "name": "useful_y", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "funny_y", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "cool_y", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "fans", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "elite", "type": "text"
                },
                {
                    "name": "average_stars", "type": "number",
                    "subtype": "float"
                },
                {
                    "name": "compliment_hot", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "compliment_more", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "compliment_profile", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "compliment_cute", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "compliment_list", "type": "number",
                    "subtype": "integer"
                },
                {
                    "name": "compliment_note", "type": "number"
                }
            ]
        }
    ]
}
```
"subtype": "integer"},
{"name": "compliment_plain", "type": "number",
"subtype": "integer"},
{"name": "compliment_cool", "type": "number",
"subtype": "integer"},
{"name": "compliment_funny", "type": "number",
"subtype": "integer"},
{"name": "compliment_writer", "type": "number",
"subtype": "integer"},
{"name": "compliment_photos", "type": "number",
"subtype": "integer"},
{"name": "name_y", "type": "text"},
{"name": "neighborhood", "type": "text"},
{"name": "address", "type": "text"},
{"name": "city", "type": "text"},
{"name": "state", "type": "text"},
{"name": "postal_code", "type": "text"},
{"name": "latitude", "type": "number",
"subtype": "float"},
{"name": "longitude", "type": "number",
"subtype": "float"},
{"name": "stars_y", "type": "number",
"subtype": "float"},
{"name": "review_count_y", "type": "number"},
{"name": "is_open", "type": "integer"},
{"name": "is_open", "type": "integer"},
{"name": "categories", "type": "text"}]
]}
}
Bibliography


