Label, Segment, Featurize: a cross domain framework for prediction engineering

James Max Kanter and Owen Gillespie
FeatureLab Inc.
Boston, MA - 02116
{max, owen.gillespie}@featurelab.co

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

***Abstract*—In this paper, we introduce "prediction engineering" as a formal step in the predictive modeling process. We define a generalizable 3 part framework — Label, Segment, Featurize (L-S-F) — to address the growing demand for predictive models. The framework provides abstractions for data scientists to customize the process to unique prediction problems.**

We describe how to apply the L-S-F framework to characteristic problems in 2 domains and demonstrate an implementation over 5 unique prediction problems defined on a dataset of crowdfunding projects from DonorsChoose.org. The results demonstrate how the L-S-F framework complements existing tools to allow us to rapidly build and evaluate 26 distinct predictive models. L-S-F enables development of models that provide value to all parties involved (donors, teachers, and people running the platform).

I. INTRODUCTION

In recent years, the data science community has experienced a sharp increase in demand for predictive models from time-driven relational data. This type of data is collected during the regular day-to-day use of physical machines (such as turbines or cars), services (such as ride sharing or an airline), and digital platforms (such as online learning or retail websites). This data has specific properties that differentiate it from images or text. It is event-driven and collected across different time scales, and contains a multitude of data types, including categorical, numeric, and textual.

Improving the process of building predictive models for such data presents immense opportunities for organizations. Past work has focused on making automated tools for training optimal machine learning models [1], and on automating feature engineering [2]. However, we contend that a more complex and human-driven process still remains to be defined and solved. This process–of defining the outcome user is interested in predicting, finding past occurrences of this outcome in the data, and forming learning segments (prior to the outcome) from which a model could be trained–is still manual, and is executed on a case-by-case basis.

Most workflows assume that data has already undergone this process and is ready for feature engineering. At the time of this writing, KAGGLE, a popular website for crowdsourcing data science, had just released a challenge in which participants build a model that can differentiate between interictal EEG segments (those occurring between seizures) and preictal ones (those occurring prior to seizure)[1]. This data was preprocessed to: separate data files of 1 hour segments labeled as preictal and interictal; make sure preictal segments ended 5 minutes before; and interictal segments were collected randomly but at least 4 hours away from seizure activity. Such data is immediately ready for feature engineering and machine learning when released. Arguably, this is one of the most important value additions KAGGLE provides.

[1]https://www.kaggle.com/c/melbourne-university-seizure-prediction

Currently, practitioners face many common challenges in preparing training examples for feature engineering and learning a predictive model, including (1) defining how the outcome to be predicted is computed and searching through variations in its definition, (2) setting one or more time points to check for the presence/absence of the outcome, (3) picking training examples based on domain-specific criteria (for example, in a medical application, only one example per patient is allowed), (4) maintaining gap between them, when multiple examples are picked from the same entity-instance (5) achieving balance in classes (for classification problems), (6) optimizing the parameters of a prediction definition, such as how far ahead to predict and how much past data to use, (7) deciding on the timespan of data to use for learning, (8) preventing label data from leaking into feature extraction, (9) handling time deltas defined in absolute time or number of samples. Across different domains, these challenges manifest themselves in several disparate ways. Thus, there exists no unified framework to enable data scientists to quickly ingest data, parametrically set up a prediction problem, and segment the data for feature engineering.

We formally define "*prediction engineering*" as the process that transforms time-driven relational data into feature vectors and labels that can be used for machine learning. It aims to abstract away common data preparation tasks prior to feature engineering, allowing data scientists to focus on the specifics of the prediction problem at hand. In this paper, we present the Label, Segment, Featurize (L-S-F) prediction engineering framework. By formalizing prediction engineering, we not only enable the rapid iteration of predictive models on a single dataset, but also unify model building in seemingly disparate domains.

Our major contributions include:

- A set of abstractions that enable profound flexibility when defining prediction problems.
- The Label, Segment, Featurize (L-S-F) prediction engineering framework, which systematically prepares relational data for machine learning.

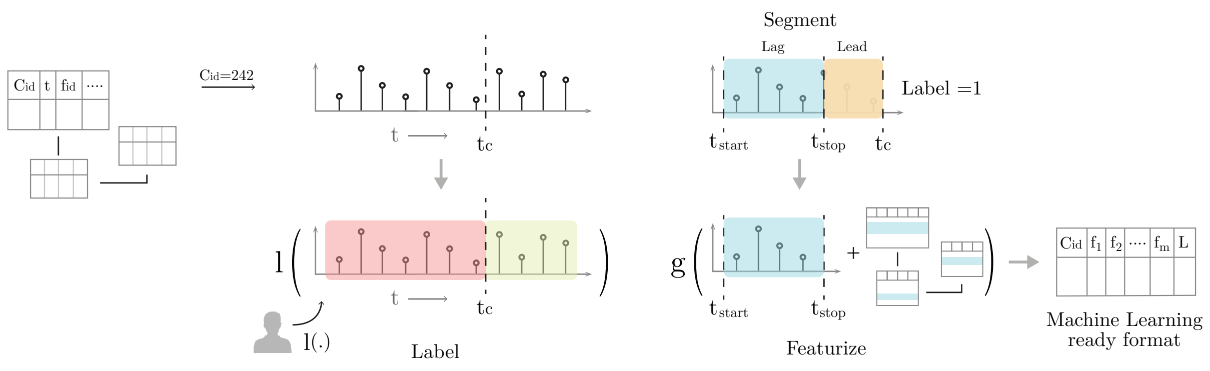

Fig. 1. An overview of the L-S-F framework for prediction engineering. In the "label" step, the user supplies a labeling functions function to compute a prediction label after a cutoff time, t_c. In the "segment" step, a *learning segment* from before t_c is identified using parameters such as *lead* and *lag*. Finally, in the "featurize" step, data from the *learning segment* is transformed into a feature vector that can be used by machine learning algorithms to build predictive models.

- Demonstration of the efficacy of the L-S-F framework on 5 unique prediction problems defined over a dataset.

The paper is organized as follows: In section II we present a motivational example, and in section III we present related work. In Section IV we present the L-S-F framework for prediction engineering. In section V, we elaborate how the output of L-S-F interacts with the feature engineering software. In section VI we apply L-S-F to problems from 2 domains, and present experimental results for the KDD-2014 dataset. In section VII we discuss the implications of L-S-F and what we learned as its first users. Section VIII concludes.

II. Motivational example

Consider a retailer that wants to predict if a given customer will churn, and to react to that prediction as early as possible. The retailer has established a specific definition of churn, and has at their disposal a timestamped data set containing all of the store's customer transactions.

To construct training examples for machine learning, the data scientist must first implement a definition of churn that identifies customers who can be used as *positive* and *negative* training examples. If we are using a time window for churn – for example, *will this customer churn in the next 6 months?* – then a given customer could be used as either a positive or negative example, depending on the time point at which we choose to label them. For each labeled customer, the data scientist identifies a segment of data prior to the labeling time point to learn a model. The data scientist performs feature engineering over this segment of data to avoid leaking information into the features.

How training examples are labeled depends on the definition of churn and when the labels are calculated, while feature extraction depends on how far in advance the data scientist wants to predict and how much historical data is used. After building a end-to-end model, altering these parameters should be easy. For instance, if it is discovered that churn can be predicted accurately 1 week in advance, it is natural to explore increasing that amount of time to 1 month. An exciting aspect of predictive modeling is imagining other *"what if"* scenarios:

- *What if* the retailer wants to change how customer churn is defined from 1 month of inactivity to 3 months?
- *What if* the retailer wants to predict churn only after the first purchase by the customer, for operational reasons?
- *What if* the data scientist wants to use the same pipeline to predict customer engagement on a *per-session* basis?
- *What if* a second retailer presents a similar business problem and data set, and wants to use this same previous work?

III. Related Work

Many researchers have studied the process of building predictive models. Like this paper, many have specifically focused on time-varying data and underscored the attention that needs to be paid when preparing such data for machine learning algorithms. For example, [4] writes of label leakage in data mining competitions. Later [5] proposes a solution to avoid leaks by defining the targets of predictive models using time-based measures.

[3] explicitly defines label leakage and highlights scenarios in which it impacts predictive modeling. The writers suggest the "learn-predict separation" paradigm to handle incoming data by tagging it with time labels in order to avoid contamination by unallowable data during feature engineering. There has been steady focus on the careful preparation of data such that data used to evaluate the target does not somehow leak into learning.

Meanwhile, [9] advocates that emphasis be paid to steps ordinarily taken outside the modeling and learning stages of a machine learning endeavor. These include phrasing a problem in machine learning terms, generating features, and interpreting and publicizing the results. With generating features being the focus of [6] and [8], and more recently [2], the goal of this paper is to systematize the process of phrasing a problem. This in turn will enable end-users to generate many problems, analyze the resulting predictive models, and disseminate them.

The L-S-F framework we develop in this paper complements existing methods for feature engineering over relational datasets. Propositionalization [6] is a method to flatten an entity for prediction using aggregations. ACORA [8]

Name	Description
e	An entity – for example, "customer."
eid	An entity instance – for example, "customer 24."
t_c	cut-off time point.
t_{start}	Starting time point for a *learning segment*.
t_{stop}	Ending time point for a *learning segment*.
l	Label id.
A	List of three tuples $< eid$, t_c, $l >$.
$L(.)$	Labeling function that contains logic for the outcome.
w_l	The time span of data after a t_c used by the labeling functions to come up with a label (time delta)
`lead`	How far ahead of time to predict the outcome (time delta).
`lag`	How much past data to use for the prediction (time delta).
eid^D	Data pertaining to a particular instance of an entity.
n_w	The upper bound on the number of t_c's to form
O	Amount of time between two consecutive t_c's when $n_w > 1$ (time delta)
`gap`	The space between two consecutive training examples from the same entity instance (time delta)
`anchor`	Time point used to anchor `lag` from. If a prediction is desired as soon as an entity instance appears in the data, this is set to START

TABLE I. Notations and definitions. Also included here are the parameters for the L-S-F framework. All time points are expressed in `epoch time` and all time deltas are experessed in `seconds`.

adds distribution-based aggregates. Deep Feature Synthesis [2] builds features by stacking functions across relationships in the dataset.

IV. Label-Segment-Featurize

As we worked through numerous data science problems across domains, we identified several underlying commonalities. In a typical situation, a data scientist building a predictive model will start by working with a written or spoken definition of the *outcome* the end user is interested in predicting, as well as constraints, such as how far ahead of time the end user wishes to make the prediction (*lead*) or how much data should be used to do so *lag*. Given this information, we divide the workflow, in which raw data is transformed into training examples, into three steps: *Label*, *Segment* and *Featurize*. In the next three subsections, we describe these three steps in detail, along with the methods we have developed for each.

A. Label

The goal of the Label step is to *traverse* through an entity-instance's data and identify a list of three tuples: $< eid, t_c, l >$, where eid identifies the entity-instance to which the training example belongs, t_c the *cut-off* time point (or index) prior to which data can be used for learning the model, and l the category of outcome (in binary cases, "positive" or "negative"). We further break this step down into two substeps:

1) defining a labeling functions and
2) traversing the data in search of labels.

Defining a labeling functions: A labeling functions, $L(.)$, contains a user-defined logic that

- is applied to the the data within the labeling window,
- can optionally use data prior to the *cut-off* t_c, but must use the data within the labeling window, and
- produces a *label* the user is interested in predicting.

When we attempted to define labeling functions for a variety of well-known predictive problems over different domains, we

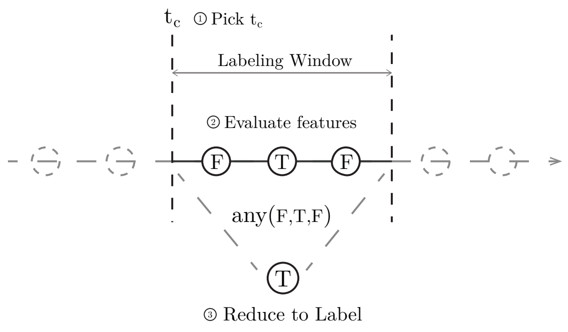

Fig. 2. The *feature-reduce* abstraction for defining prediction labels. It works by calculating a feature value for each event/sample in the labeling window. In this example, we are calculating a binary feature at each event (purchase event or not). Then, it reduces that list of features to a single label using a function such as ANY

realized that they typically share two additional characteristics. First, samples within the window are each transformed into another value, and in almost every case, this value is equivalent to a *feature* in the machine learning sense. Second, an aggregation logic is applied to these values in order to generate the label.

Noting these consistencies, we created a *feature-reduce* abstraction. This abstraction allows data scientists to develop their own labeling functions with the same language they will later use to develop features. Examples of how *feature-reduce* is applied in 3 different problems in 3 different domains can be seen in Table II.

Traversal algorithm: Given the labeling functions, our next goal is to design a traversal algorithm that searches through the data to identify several possible training examples. A simple design of such an algorithm would entail setting t_c at every time point where a data sample is present in the entity-instance's time span, and evaluating the labeling functions. To do this in a tractable way, we propose to iterate over the data

Outcome	Feature	Reduce	Window
At least one purchase in the next week	`Compare("event_type","=","purchase")`	any	7 days
Rolling average temperature greater than 100 degrees over last 10 samples	`mean = RollingMean("temp",(10, "samples"))` `Compare(mean, ">", 100)`	all	10 observations
Positive bank account balance for next 30 days	`Compare("balance", ">", 0)`	all	30 days

TABLE II. THREE EXAMPLES FROM DIFFERENT DOMAINS OF HOW TO DEFINE PREDICTION PROBLEMS USING FEATURE-REDUCE IN THE L-S-F FRAMEWORK BY SPECIFYING A (FEATURE), REDUCE FUNCTION, AND PREDICTION WINDOW

Algorithm 1 Traversal algorithm

```
1:  procedure TRAVERSE($L(.)$, $O$, $w_l$, $n_w$, $eid^D$, $eid$, $A$)
2:      $eid^D$: Data from an entity instance $eid$.
3:      $t_s \leftarrow starttime(eid)$
4:      $T_c \leftarrow$ SET-CUTOFF-TIMES $(O, t_s, n_w)$
5:      for $t_c \in T_c$ do
6:          $D \leftarrow eid^D[t_s : t_c + w_l]$
7:
8:          $l \leftarrow L(D)$
9:          $A \cup \{ eid, t_c, l \}$
10:     return $A$
11:
12: procedure SET-CUTOFF-TIMES($O$, $t_s$, $n_w$)
13:     $T_c[1] = t_s$
14:     for $i = 2$ to $n_w$ do
15:         $T_c[i] = (T_c[i-1] + O)$
16:     return $\{ T_c \}$
```

Algorithm 2 Segmentation algorithm

```
1:  procedure SEGMENT(A, $lag$, $lead$, $anchor$, $gap$, $multi$, $randomize$)
2:      S={ }
3:      for $id \in eid$ do
4:          $t_s \leftarrow starttime(id)$
5:          $\bar{T} \leftarrow t_c$ where $eid = id$ and $t_c > t_s + lead + lag$
6:          $\bar{l} \leftarrow l$ where $eid = id$ and $t_c > t_s + lead + lag$
7:          if $anchor$ = 'START' then
8:              $t_{stop} = t_s + lag$
9:              return $S \cup \{eid, t_s, t_{stop}, \bar{l}[1]\}$
10:         else if $multi$ = 'FALSE' then
11:             if $randomize$ = 'TRUE' then
12:                 $index \leftarrow rand(len(T))$
13:                 $t \leftarrow \bar{T}[index]$
14:                 $l \leftarrow \bar{l}[index]$
15:                 $S \cup$ FORMSEGMENT($eid$, $t$, $lead$, $lag$, $l$)
16:             else if $randomize$ = 'FALSE' then
17:                 $t \leftarrow \bar{T}[1]$
18:                 $l \leftarrow \bar{l}[1]$
19:                 $S \cup$ FORMSEGMENT($eid$, $t$, $lead$, $lag$, $l$)
20:         else if $multi$ = 'TRUE' then
21:             $q \leftarrow t_s + lead + lag$
22:             for $t \in T$ do
23:                 if $t > q$ then
24:                     $index \leftarrow$ FIND$(T == t)$
25:                     $l \leftarrow \bar{l}[index]$
26:                     $S \cup$ FORMSEGMENT($eid$,$t$, $lead$, $lag$, $l$)
27:                     $q \leftarrow t + gap + lag + lead$
28:                 else $S \cup \{ \}$
29:
30: procedure FORMSEGMENT($eid$, $t$, $lead$, $lag$, $l$)
31:     $t_{stop} \leftarrow t - lead$
32:     $t_{start} \leftarrow t_{stop} - lag$
33:     return $\{eid, t_{start}, t_{stop}, l\}$
```

samples by specifying the number of t_c's to consider, n_w, the spacing between them, O, and the size of the labeling window, w_l. Pseudocode is presented in Algorithm 1, and the process is illustrated through an example retail dataset in Figure 3. It is important to note that the traversal algorithm does not make use of any information about the logic inside the labeling functions. The only information that is shared between it and the labeling functions is the length of labeling window. We imagine that several different and perhaps efficient alternatives are possible for the traversal algorithm. To preserve generalizability, we recommend alternate designs to follow the abstractions for the labeling function, and the inputs and outputs, we suggest for the algorithm. For example, an alternate algorithm can use the size of labeling window to skip portions of time series after one instance label is found. While using information from the logic inside the labeling function can aid in an efficient traversal, it could potentially result in a very specific traversal algorithm.

B. Segment

Given the *lead* and *lag* parameters and the labels, this step identifies segments defined by the four tuples $< eid, t_{start}, t_{stop}, l >$, where t_{start} and t_{stop} represent the time points that will bound the samples used for learning. We call this the "*learning segment.*"

Lead, Lag, and units

- **Lead**: For a predictive problem, the $lead$ time specifies how far ahead the user wants to predict the label. For many predictive problems, predicting far ahead (high $lead$) enables value creation, and makes predictive problems interesting. For example, a retailer may want to predict future customer churn as soon as possible.
- **Lag**: Next, we must decide how much historical data we wish to use to predict the label. This value is known as the *lag*. Numerous considerations are taken into account to pick this value, including memory and computational constraints, storage, and the amount of time that has passed since the last important event. We also pick a point to calculate the lag relative to called the *anchor*. Typically, there are three options for setting the anchor: the first time or data point for the entity-instance t_s, the point in time $t_c - lead$, or a fixed time point in the entity-instance's

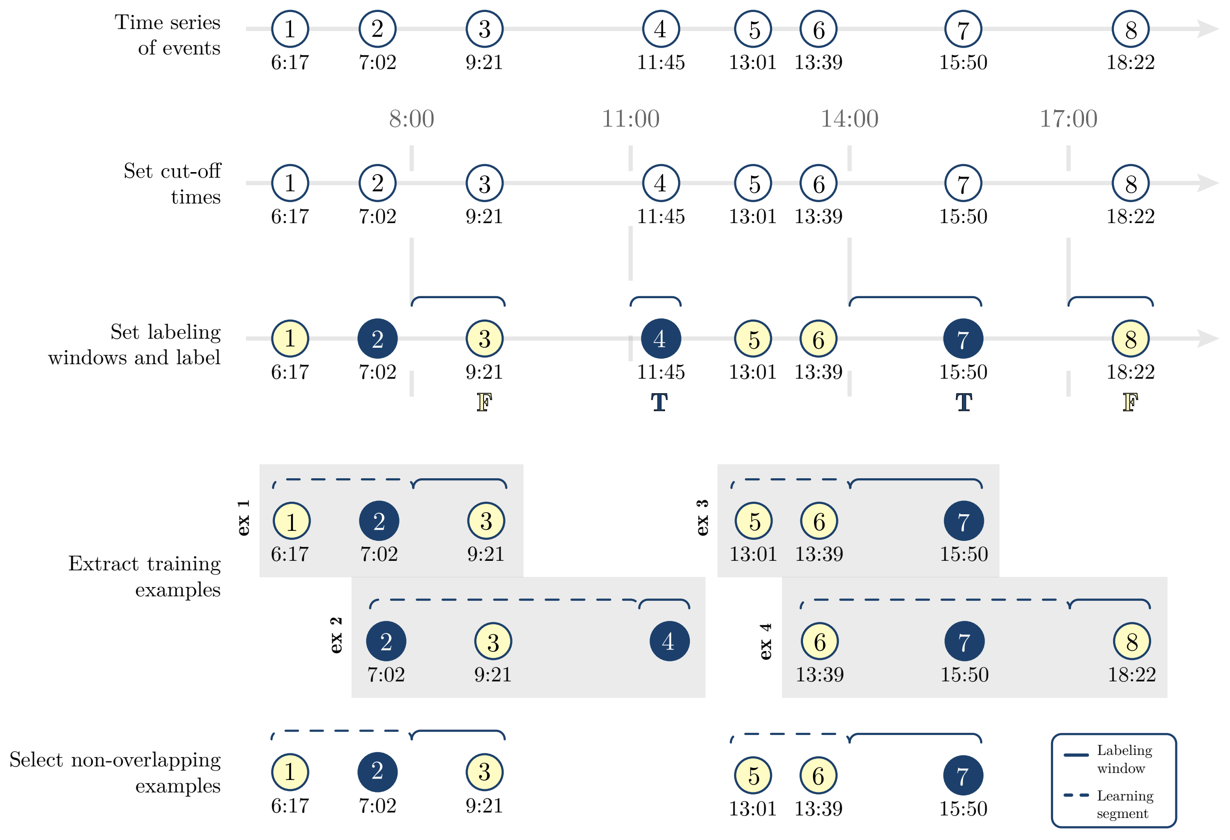

Fig. 3. Figure illustrating the process of traversing, labeling, and segmenting the data to generate training examples for feature engineering. An event time series from customer transactions data base is chosen as an example. Each node represents a customer transaction and colors represent different types of events - *blue- purchase*, *yellow - browse*. Exact time stamps are shown below the node. For this customers data, it sets 4 t_c's. The labeling window is set as 1 event and the outcome of interest is *whether or not a customer makes a purchase*. In training examples are extracted based $lag = 2$ events, and $lead = 0$. Finally, 2 non-overlapping training examples are selected (gap in this case is set to zero).

lifespan.

- **Units**: For both *lead* and *lag*, we specify the amount of past data in absolute units- seconds. We allow users to specify in relative units-the last 10 samples or events. When relative units are specified a function transforms them into time delta in seconds by scanning the data.

Considerations for selecting from labeled examples: After the label step, a given entity-instance may lead to multiple occurrences of the outcome of interest. For example, a customer may have bought a particular product multiple times, or a patient may have several seizure events. In such cases, the user must decide whether to extract multiple training examples, and, if they choose to extract just one, which one they should pick. Below, we walk through these choices, and the considerations they entail.

- Picking multiple examples: The user may choose to extract training examples (and their corresponding learning segments) from a single entity-instance's data. In cases where not many instances of the entity exist, this is a valuable way to maximize training examples. In particular, the user may wish to extract non-overlapping learning segments using the *gap* parameter. Two learning segments are considered non-overlapping when the segments between their t_{start} and t_{stop} points do not overlap. In order to extract multiple exemplars, the segmentation algorithm must iterate over the labeled examples to find non-overlapping segments where the *gap* constraint is satisfied. In algorithm 2, we present a greedy approach to extracting non-overlapping learning segments.
- Picking a single example: When the user chooses to create only one example from each entity-instance, a natural question is: which one? Two possible choices are:
  - ○ First: the user may choose the first occurrence.
  - ○ Random: the user may pick an example randomly. [2]
- Maintaining class balance: When there are multiple examples from the same entity instance, both positive and negative, we choose those that help maintain the balance.

Segmentation algorithm: With *lead* and *lag* defined, and either single or multiple training examples chosen, we present a greedy approach to form *learning segment* in Algorithm 2. It is important to note that the segmentation algorithm does not use any information from the or the traversal algorithm other then the list of labeled exemplars. Unlike traversal algorithm, it does not operate on the actual data. Similar to the traversal algorithm, we see different ways to design the segmentation algorithm.

[2]This allows a possible feature that counts the number of past occurrences of an outcome, which could be predictive of the outcome in future.

V. Feature Engineering

In the previous step, we selected a subset of the training examples and determined the *learning segment* to use to build features for them. Our next step is to compute features and produce a matrix ready for machine learning. To compute features using only *allowable* data, we can approach feature engineering via two methods. The first method involves manipulating the data before passing it on to the feature engineering software. In the second, we propose changes to core mathematical operations utilized to compute features.

Manipulating data: In this approach, to extract features for each training example in S, we use the corresponding t_{start} and t_{stop} and remove all observations for any entity whose time stamps are not within those two boundaries. Thus the feature engineering software is ”unaware” of any data that exists beyond those time points. The obvious advantage of this approach is that the user does not have to change their feature engineering software; however, this method can require significant computational effort in identifying the observations, which could be split across multiple tables or do not fall within those timestamps, as well as memory to create a copy of the subset of the data.

Manipulating operators: Alternately, we propose to rethink the typical input-output of the mathematical functions used to engineer features. Feature engineering relies on mathematical operators such as max, min, $average$, among many others. Typically these functions take an *array* of values and output a single value, $f \leftarrow g(\bar{a})$. Instead, to account for using only *allowable* data, we modify these functions as $f, t \leftarrow g_b(\bar{a}, \bar{t_a}, t_{start}, t_{stop})$, where $\bar{t_a}$ is an *array* of time stamps that correspond to the observations in $\bar{a}$, f is the computed output of the function, and t corresponds to the $maximum$ time stamp of any observation in the $\bar{a}$ used to compute the feature. This new function modifies the original $g(.)$ function as follows:

Algorithm 3 Bounded feature functions
```
1: procedure $g_b(\bar{a}, \bar{t_a}, t_{start}, t_{stop})$
2:     $indices \leftarrow t_{start} < \bar{t_a} < t_{stop}$
3:     $\bar{a_b} \leftarrow a[indices]$
4:     $f = g(\bar{a_b})$
5:     $t = max(\bar{t_a}[indices])$
6:     return $f$, $t$
```

Thus we redefine the feature functions in any of the generic feature extraction software [2].

VI. Experimental results

We begin this section by presenting how L-S-F generalizes. To illustrate this, we first pick two different domains, present a generic data model for each, and demonstrate how users can set up prediction problems by simply tweaking a few parameters. Next, we pick a specific data set and demonstrate how L-S-F framework can help users explore multiple problems, but investigate impact of different parameters on the predictive models accuracy.

Tables	Fields
Online retail	
Customers	C_{id}, $dc_{i,\dots,k}$, $xc_{i,\dots,k}$
Logs	T, C_{id}, $dl_{i,\dots,k}$, $xl_{i,\dots,k}$
$meta_i$	dc_i, val
Sensors	
Components	Co_{id}, $dc_{i,\dots,k}$, $xc_{i,\dots,k}$
Logs	T, Co_{id}, $dl_{i,\dots,k}$, $xl_{i,\dots,k}$
$meta_i$	dc_i, val

TABLE III. Generic data models in three different domains.

A. Generalizability across domains

The typical data model contains the entity of most interest to data scientists, a transactions table with observations over time, and optionally one or more tables with meta-information.

Online retail: In this domain, the entity over which most prediction problems are defined is the *customer*. The "customer" table can have several discrete (ordinal or categorical) variables and numeric variables to begin with, and there is a logs table with every interaction any customer has made with platform. Each entry has a time stamp T and a customer id, identifying the customer to whom that line belongs. There are typically several tables containing meta-information. When presented with dataset in this model, we are able to define numerous predictive problems by simply setting the *feature* function to be:

```
Compare($dc_i$,"=",val)
```

for any of the discrete/categorical variables and

```
y = rollagg($xc_i$,($n$, "samples"))
Compare(y, ">", $th$)
```

for any continuous variables, where $rollagg$ is an aggregation function like *rolling mean* and n is number of samples, th is the threshold. Then one can specify a *reduce* function over a labeling window. We present an example of a problem that could be defined over this type of data, in the first example in Table II. In that example, the categorical variable dc_i = `event_type` and val corresponds to `purchase`.

Once the data is ingested in the L-S-F framework, a user can explore numerous prediction problems, by simply specifying parameters for these functions, changing the *reduce* functions as necessary and setting parameters for *traversal* and *segmentation*.

Sensors data: In this domain, data is collected from sensors, which record observations about a component or phenomenon. Each sensor's data is usually aggregated over a period of time using several aggregation functions (mean, standard deviation and others), and the resulting data is provided as numeric values in the logs table. The time stamps in T are regular. Discrete values are often stored, which capture the settings and/or state of the component during that time interval. Tables carrying meta-information about the sensors, the component, and other variables may also be present. Similar to the previous example, we can specify a *feature* function on any of them continuous variables as:

Experiment	Feature	Reduce	Window
P1: Reach funding goal	`CumSum("donation_amount") >= Project.funding_goal`	any	∞
P2: Reach funding goal in x days	`CumSum("donation_amount") >= Project.funding_goal`	any	30, 60, 90 days
P3: x percent of donation over y dollars	`num_donations = CumCount("donations")` `num_large= CumCount("donations", where="donation_total>y")` `num_donations/num_large > x`	all	∞
P4: All donations have tip	`num_donations = CumCount("donations")` `num_tips = CumCount("donations", where="included_optional_support=1")` `num_tips / num_donations == 1`	all	∞
P5: More than x donations	`CumCount("donations") > x`	any	∞

TABLE IV. ALL FIVE PREDICTIVE PROBLEMS OVER KDD-2014 DATA DEFINED USING *feature-reduce* ABSTRACTION. TOGETHER WITH THESE *feature-reduce* DEFINITION AND TABLE VI, ALL FIVE PROBLEMS ARE DEFINED IN THE L-S-F FRAMEWORK.

```
y = rollagg("xc_i",(ns, "samples"))
Compare(y, ">", th)
```

We present an example of a problem that could be defined over this type of data, in the second example in Table II. In that example, the continuous variable xc_i = `temp` and th corresponds to 100. Similar to above, users can explore many predictive problems by systematically specifying the different *feature* and *reduce* functions.

B. Exploring predictive models

The two foundational claims we make for the L-S-F prediction engineering framework are:

– It allows users to switch between predictive problem definitions by simply changing a few lines of code.
– By simultaneously abstracting common data preparation tasks away from the user and enabling them to tweak the parameters of the predictive problem definition, it enables them to focus on exploring variations in the predictive problem definition, which could generate immense value. Table V highlights some common variations.

In this section, we choose a well-known, publicly available data set from KDD-2014, and demonstrate how we can easily set multiple prediction problems (see Table IV) and explore several variations of those problems by simply tweaking parameters in the L-S-F framework.

Exploratory question
How far ahead can I predict an outcome?: Users may want to change the *lead* parameter to see if they can predict an outcome sooner rather than later.
How sensitive is the predictive accuracy to a parameter in my problem definition?: Often, users flexible with values for the parameters for the prediction problem itself.
How much historical data is needed to predict the outcome?: The user can control prediction time by setting the *lag* parameter to the desired time when the *anchor* is set to *start*.

TABLE V. THREE EXAMPLES OF EXPLORATORY QUESTIONS USERS CAN ASK WITH THE PREDICTION ENGINEERING FRAMEWORK IN PLACE THAT ULTIMATELY AIDS IN DERIVING VALUE OUT OF THE DATA.

	Parameter	Parameter choice
P1: Reach funding goal	Lag	1 3 5 7 days
P2: Reach funding within	Lag	3 6 9 days
	y days	30 60 90
P3: x% of donations over y dollars	x%, y	[> 30%, \$> 100], [> 35%, \$< 10]
	Lag	3 6 9 days
P4: All donations have tip	Lag	3 6 9 days
P5: More than x donations	x	10
	Lag	3 6 9 days

TABLE VI. PARAMETERS FOR EACH PREDICTION PROBLEM AND THEIR POSSIBLE VALUES

Dataset: The KDD-2014 data set we used for our experiment comprises the metadata, resource requests, and donation history from the 50,000 most recent projects before 2014. The competition defined the related prediction problem as follows: *Using past projects' histories on DonorsChoose.org, predict if a crowd-funded project is "exciting"*. For the purposes of demonstrating the efficacy of L-S-F, we choose to define multiple prediction problems over fields that varied with time, such as a project's donation history.

Table IV presents 5 prediction problems we explored in this dataset and how they are defined using feature-reduce. The questions in Table V translate to the parameters presented in Table VI. The parameters were chosen after we conducted an initial exploration of the data using the distributions in Figure 4. To build a feature matrix, we used the Deep Feature Synthesis algorithm[3]. With the feature matrix and labels, we trained a random forest and applied 10-fold cross validation[4].

In the following subsections, we present the rationale behind the problems we selected, as well as their parameters. Results for all prediction problems and parameter settings are displayed in Figure 5.

P1: Predict whether a project will reach its funding goal: For this problem, we predict whether or not a project will

[3]Implementation provided by Featuretools (docs.featuretools.com)
[4]Implementation provided by scikit-learn [7]

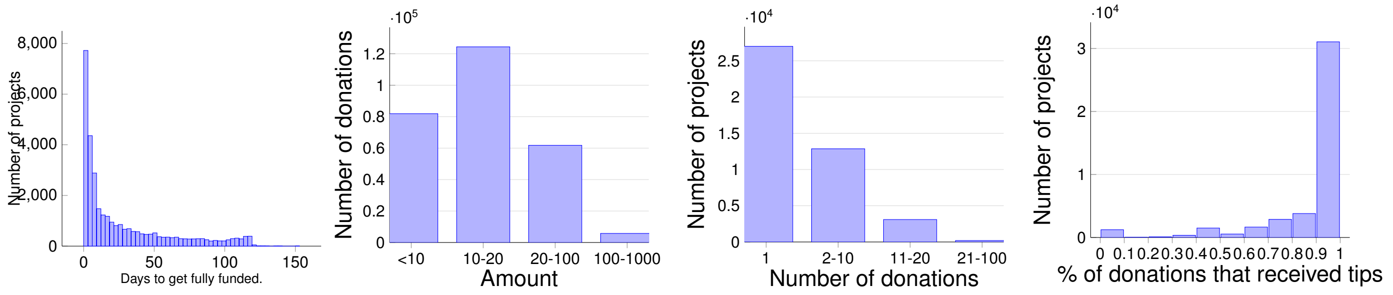

Fig. 4. Distributions that were used to inform the paramters to use for the each of the prediction problems

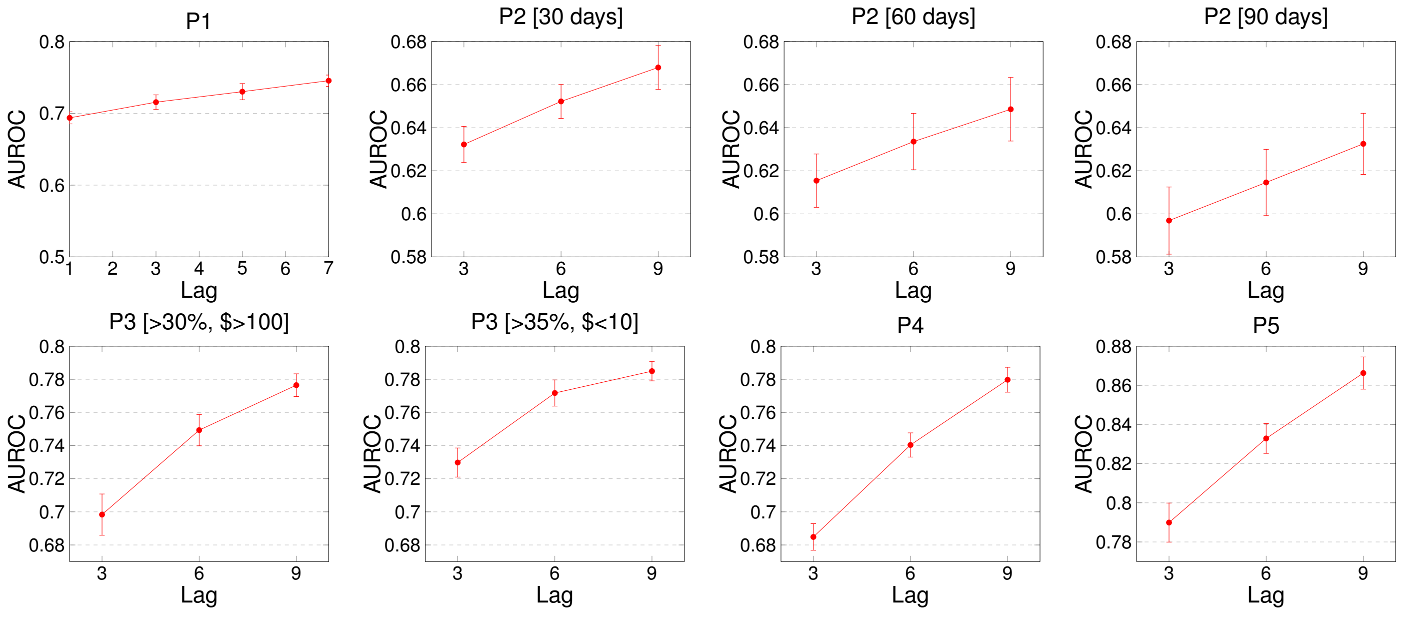

Fig. 5. The AUCs achieved as the *lag* parameter is varied for different prediction problems. The error bars represent the standard deviation across the folds of cross validation. While increasing the amount of past data increases prediction accuracy in all cases, the magnitude of the increases and error bound vary widely.

reach its funding goal within the allotted time, as soon as a project is posted to DonorsChoose. We set the goal as 100% of requested funds. Out of the 50,000 projects we chose for this study, 17,546 projects do not reach their funding goal, even after 150 days. Because we aim to be able to predict this outcome as soon as the project is posted to the website, we choose the anchor to be the starting point, and vary the *lag* from 1 day to 9 days. To be able to use a total *lag* of 9 days and keep the same projects for training, we remove all the projects that were funded within the first 10 days.

P2: Predict whether a project will reach its funding goal in x days: A natural follow-up to the previous question is to ask whether the project will reach its funding goal within a certain number of days. This may be helpful for teachers, who could then plan for when they will have the funds in hand.

We see that a significant proportion of the projects were funded within 30 days, with smaller portions funded within 60 days and 90 days, making them reasonable parameter choices. We would like to be able to predict these funding windows within 3, 6 or 9 days of the proposal being posted.

P3: Predict whether a project has x% of its donations over or under y dollars: The next question we target is whether a project attracts a lot of small donations (“grassroots”), or a significant proportion of large donations (“big donors”). This sort of labeling could be valuable to DonorsChoose.org, who could use it to to match donors with projects that fit their donation style.

Using the distribution of individual donation amounts across all projects, we see that many donations are under $10 (30% of total donations), and a relatively smaller portion are above $100 (2% of total donations). To qualify as a project that invokes grassroots-level interest, we set the donation threshold at $10, and classify the projects where 30% of donations received were under this amount. To classify the projects that have a potential to attract big donors, we set the donation threshold to be $100, and require at least 35% of the donations to be above this amount.

P4: Predict whether a project will receive tips with donations: By default, DonorsChoose.org adds a 15% tip to each donation unless the donor unchecks a box. As a result, most donations come with a tip. For most projects, this number is *all*, which translates to a value of 1. For a very small portion of projects, however, some number of donations come without tips. In this prediction problem, as soon as a project is posted, we attempt to predict whether or not the donations it garners will all have tips.

P5: Predict whether a project will receive more then 10 donations: Finally, we attempt to predict whether a project will receive more then a certain number of donations, which is indicative of its popularity. Companies doing matching offers or other forms of corporate outreach are interested in engaging with as many people as possible, so they may want to focus their attention on these projects. We observed that most projects receive 10 donations or fewer, and that a small portion receive more then that. Again, we attempt to predict this with a lag of 3, 6, or 9 days.

VII. Discussion

Towards productivity gains for data scientists: Our results show how a data scientist can rapidly formulate and solve prediction problems to respond to an increased demand for predictive models.

The productivity gains from L-S-F allowed us to answer a number of interesting questions:

- **What was the hardest problem?** Predicting whether a project would be fully funded within 90 days was the most difficult problem. Even taking into account the donations during the first 9 days did not improve the AUC.
- **Which problem would be most helped by waiting longer before making a prediction?** Predicting whether a project would garner tips alongside all its donations can gain a significant boost in AUC by taking into account the observations as they come in.
- **Which problem achieved the highest AUC?** Predicting whether a project would become popular by exceeding a certain number of donations achieved the highest AUC.

The ability to answer exploratory questions is vital for addressing the appetite for predictive models by stakeholders in any organization. In this case, *donors*, *teachers*, and DonorsChoose.org itself may each have a different objective. For example, a donor may be interested in knowing how their donation affects a project early in its funding campaign (P1 and P2), while DonorsChoose may be interested in maximizing user engagement with their website by featuring certain projects (P4). A teacher may be interested in how different actions affect the type of donor attracted to a campaign, or its popularity (P3 and P5).

Through L-S-F, we are able to accommodate all of these stakeholders. Now, switching between prediction problems requires users to write just one concise labeling functions function for the label step, rather than reworking the rest of the system. In problems P2 and P3, we even define this function with parameters to aid in exploration. Likewise, we parametrize the assemblage of learning segments for feature engineering, in order to enable iteration over important prediction options.

All 5 prediction problems use this flexibility to explore different options for the lag parameter. Figure 5 demonstrates how the impact of lag varies greatly from problem to problem. Understanding this impact with empirical data helps data scientists and stakeholders make decisions with more complete information.

Importantly, these results show how the L-S-F framework complements existing tools without overlapping with the functionality of those tools. For our experiments, we easily integrated with external libraries for for feature engineering, machine learning, and model evaluation.

Towards a generalizable, structured process: One of our fundamental contributions is giving structure to a process that is currently executed in an ad-hoc fashion.

The heterogeneity inherent in building predictive models presented the greatest challenge to developing this general structure. As we encountered disparate needs and edge cases across domains, we were forced to discard a number of complete designs, and to repeatedly come up with new abstractions to ensure that every application and scenario was covered.

Ultimately, this led us to separate the prediction engineering process into three steps: Label, Segment and Featurize. This separation, while intuitive, is also well-reasoned, and allows for assimilation of the multitude of scenarios in which data presents itself. For example, when data is already labeled, users may only want to use Segment and Featurize. This would be the case if one wanted to use this framework for the recent competition form, KAGGLE, mentioned in the introduction. Additionally, we imagine that these three steps could be carried out by different stakeholders within a domain. For example, Label could be developed by domain experts, while Segment and Featurize could be executed by data scientists.

Underneath each of those steps, we defined abstractions, as well as a minimal set of parameters that can service the heterogeneity. These include establishing the characteristics of the labeling functions functions, using a window of data for *labeling*, defining a cut-off time point t_c, and providing parameters such as *anchor*, *gap*, and *multi*, in addition to the obvious *lead* and *lag*.

Given these well-defined abstractions and parameters, we defined two algorithms, the `Traversal` algorithm and the `Segmentation` algorithm. Although we provide our implementations, we imagine that better and efficient approaches will emerge for each.

VIII. Conclusion

The L-S-F framework makes it possible for data scientists to build predictive models more effectively. It accomplishes this through clear abstractions that isolate parameters of the predictive modelling process. Our experience demonstrates that L-S-F solves common challenges, and allows the data scientist to focus on how to best achieve a goal using a predictive model rather than the underlying machinery. We conclude that prediction engineering frameworks like L-S-F have the ability to change how data scientists build predictive models.

References

[1] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[2] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. IEEE International Conference on*, pages 1–10. IEEE, 2015.

[3] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):15, 2012.

[4] Ron Kohavi, Carla E Brodley, Brian Frasca, Llew Mason, and Zijian Zheng. Kdd-cup 2000 organizers' report: Peeling the onion. *ACM SIGKDD Explorations Newsletter*, 2(2):86–93, 2000.

[5] Ron Kohavi, Llew Mason, Rajesh Parekh, and Zijian Zheng. Lessons and challenges from mining retail e-commerce data. *Machine Learning*, 57(1-2):83–113, 2004.

[6] Stefan Kramer, Nada Lavrač, and Peter Flach. *Propositionalization approaches to relational data mining*. Springer, 2001.

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

[8] Claudia Perlich and Foster Provost. Distribution-based aggregation for relational learning with identifier attributes. *Machine Learning*, 62(1-2):65–105, 2006.

[9] Kiri Wagstaff. Machine learning that matters. *arXiv preprint arXiv:1206.4656*, 2012.