The AI Project Manager

Benjamin Schreck[1]
Feature Labs Inc., Boston, MA 02116

Shankar Mallapur, Sarvesh Damle, Nitin John James, Sanjeev Vohra, Rajendra Prasad[2]
Accenture, Bangalore, India

Kalyan Veeramachaneni[3]
MIT LIDS, Cambridge, MA 02139

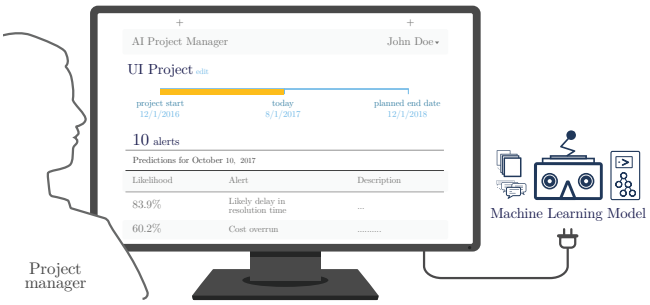

Figure 1: An augmentation tool to the human project manager is delivered via a user interface and is powered by machine learning. Once a project manager logs in to the system they get predictions 4 weeks out about different projects for which they are responsible. After selecting the project "UI Project", the user can see "alerts" based on the predictions generated by a deployed machine learning model. The user then has the ability to study alerts and add notes in the description section. This forms an artificially intelligent project manager that augments the human project manager.

1 Introduction

Across industries, highly competitive businesses continue to rely on technological innovations to help them trim costs, increase productivity, stay agile, and develop new offerings on rather short notice. One business model that has become prominent involves the use of technology service providers for the development and maintenance of software applications. Specifically, a business (*"client"*) may hire a technology service provider (*"provider"*) to develop and subsequently maintain a software application.

To address the *client*'s needs, the *provider* provides a diverse set of people with varied skill sets. These teams are distributed across the globe, and often communicate asynchronously, filling out reports focused on what is happening in their local view. This often translates to hundreds of reports covering the *provider*'s portfolio of projects, each with dozens of fields. The task of sorting through this growing volume of data, and making decisions based on it, falls on the shoulders of the *project manager*. For the particular case in question here, we had data available for training from a total of 1 762 projects. The total number of reports across all projects was 438 580, averaging out to roughly 249 reports per project, and the total number of recorded values in these reports was 40 389 171, for an average of 22 922 per project.

Typically, *project manager*s manage multiple projects. They may end up reacting to problems after the fact, performing postmortems to determine the root cause. Our team thought that a predictive solution could help the *project manager* and his/her teams. With the aforementioned data readily available to us, our mandate was simple: "Can we create a machine learning model that uses the time-varying data to identify overarching patterns and predict critical problems ahead of time?" These predictions would enable the project manager to selectively focus on a subset of projects, anticipate the occurrence of critical problems, determine the nature of the problems, pinpoint the areas that would be impacted, and take remedial action.

[1] ben.schreck@featurelabs.com
[2] {s.mallapur, sarvesh.m.damle, nitin.john.james, sanjeev.vohra, rajendra.t.prasad}@accenture.com
[3] kalyanv@mit.edu

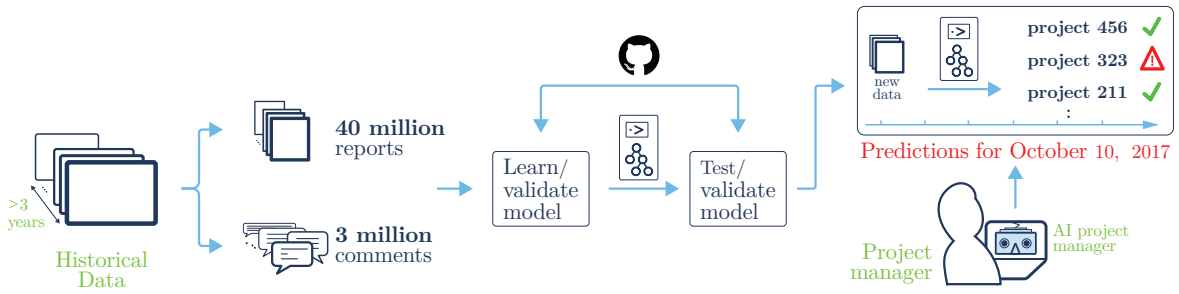

Figure 2: The process of creating an artificially intelligent project manager. Two distinct steps enable it - learn and validate model in a training phase, test and validate model in production (deployment phase). The same software and the same `APIs` are used in both phases.

The "AI Project Manager": We thus embarked on creating a system dubbed the "*AI Project Manager*". After the *project manager* logs into this system through a user interface, he can ask to see predictions for those projects that are under his/her purview. These predictions are generated remotely and served to the UI, which acts as an augmentation tool for *project manager*s. A suite of machine learning models that can predict different outcomes ahead of time powers the system, and these models are continuously updated, tested and deployed. Figure 2 shows the process that creates the models and their interaction with the *project manager*.

First model: predict delivery metrics 4 weeks ahead: The delivery metrics provide a transparent way to quantitatively measure outcomes and quality of service. For example, one metric may require that a software bug be resolved within 4 hours of its initial identification. The bugs are raised as tickets, and their responses and the subsequent resolution are all logged and time-stamped. Thus, the amount of time it took to resolve the problem can be calculated, and the number of times this metric was met (or not) can be quantified.

As an added complexity, delivery metrics can vary in terms of criticality. For example, it is critical that banking or retail applications be available to customers 24×7, with an up-time guarantee of 99.9 percent. By contrast, providing weekly reports regarding overall sales or customer abandonment may be less critical. The first predictive model we built and deployed answers the general question *"What is likelihood of not meeting a requirement for a metric"*, for metrics of varying criticality determined by both *client* and *provider*, and does so 4 weeks ahead.

Through this work, we make the following contributions:

The AI project manager: Our predictor has been deployed since September 2017, and has served predictions so far. These predictions are across projects and are made on a weekly basis. Our performance has remained steady through these weeks.

An agile system to develop, test and deploy machine learning models: We present the first use case of our new paradigm, **ML2.0**[1], which focuses on the continuous development, testing and deployment of machine learning models. This marks a significant departure from how applied machine learning is usually practiced: learning and discovering the model in a exploratory phase, and re-engineering the software to deploy the model.

Automation tools to support ML2.0: Our data science automation tools – such as `Featuretools` and others – have made this possible. Our biggest achievement is the usage of the same software tools and `APIs` to learn and deploy the model, which saves significant costs in operationalizing machine learning.

2 Building the AI project manager: ML2.0

A tool for predicting the likelihood that a required metric will not be met is just one of the many predictors that our team wants to develop, test and deploy as part of the AI project manager system. As researchers ourselves, we have undertaken many projects that involved engaging with a dataset by asking the question: "*Is it possible to predict the outcome of interest?*" As a result, our workflows were oriented toward the comprehensive analysis required to answer that question. Once we finished this analysis, we presented it, suggested which model worked, and proposed, with confidence, that the model be put to use. This traditional tack had two drawbacks: one, it separated *subject matter expert*s (in our case, the *project manager*s) from the process, and two, once the workflow was done, a massive re-engineering effort was required to actually deploy the model.

In this paper, we are instead following a different paradigm,

[1] A paper accompanying this article describes this new paradigm.

called **ML 2.0**. **ML 2.0** provides a comprehensive, structured process, and when coupled with data science automation technologies, enables *subject matter experts* and developers to develop and deploy machine learning models while remaining as comprehensive as any of our previous academic research papers/projects. Figure 4 shows the steps involved in the **ML 2.0** workflow. These steps include ingesting data and forming a structured representation, prediction engineering (to formally specify the outcome of interest), feature engineering (to generate features that can be fed into the machine learning model), and modeling and operationalization steps, to learn the model and make the necessary transformations to its output to generate real-time predictions. These steps are supported by automation tools that did not exist even 2 years ago. These tools enable (a) adaptation of the process based on subject matter knowledge, (b) searching and tuning functionality to create the best possible solution, and (c) validation and testing at different stages in the process. In Sections 4 to 8 we describe the end-to-end process, and how simple calls to our automated tools make all of this happen.

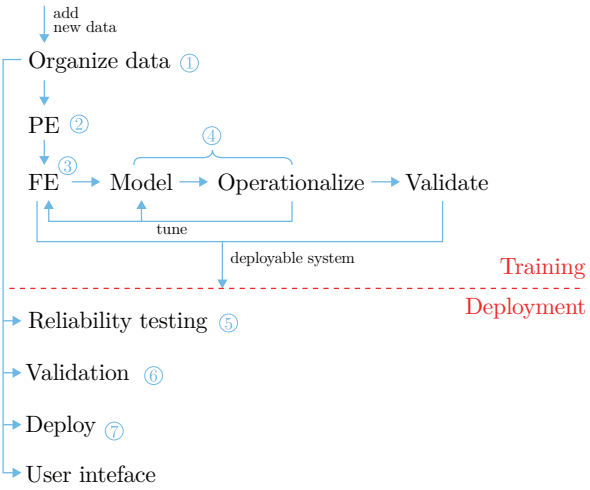

Figure 4: The ML2.0 workflow. The structured process enables a developer or a domain expert go from raw data to serving predictions in a user interface. The process is broken down into steps, allowing users to insert their subject matter knowledge, validate and test at different stages of the process. With our automation tools, this entire workflow can now be done by a single team. PE ← prediction engineering, FE ← feature engineering.

3 Prior work

There have been several exploratory attempts to predict risk factors for large-scale managed projects. Ling et al. (Ling & Liu, 2004) were able to predict the performance of managed construction projects using recorded metrics with some overlap to those that were recorded in our managed software project data. Hu et al. (Hu et al., 2007) determined it was possible to evaluate the risk of failure for software projects using results from interview questions asked of the managers.

4 Data warehouse to forming an *Entityset*

For our first model, we focus on the *maintenance* phase of each software application's life cycle. Applications enter this phase after the initial development work on them has been completed, and the primary job of the *provider* is to maintain proper functionality for the software application that has been developed. To meet this goal, project managers record hundreds of metrics about each project on a weekly or monthly basis in several different types of reports.

The first step in building any machine learning model is choosing which data to use. This is perhaps the most manual process in our workflow. In this case, the team went to the data warehouse and completed the following steps:

1. **Extract relevant tables** The full database contained 11 tables. For our purposes, 6 relevant tables were chosen.

2. **Filter data by time** The data had been recorded differently prior to 2012. We left out data recorded after 2016 so that the deployment team could test our model on data from that time period.

3. **Remove unnecessary columns** We worked with the *subject matter experts* to identify 50 columns that did not contain any useful information, either because very few projects actually had recorded information for those fields, or because the fields themselves described irrelevant information.

Data Fields Categorization: For the purpose of extracting features (patterns), fields must be organized into two broad categories: *static* or *time-varying*. Fields that do not change over time were placed in the static *Project Attributes* table (described in Section 4.1). Fields that do change over time were put into multiple tables using an *Entityset* representation (described below).

Static fields: Attributes such as the name of the project, its primary location, the person or division in charge of it, or when it started.

Time-varying fields

– **Attributes** that may change as the project progresses in time, such as currently assigned project managers, primary technologies used, the current budget, the number of full-time employees allocated, or the current phase of the development.

```
es_modeling["Project Attributes"].index
>>> "project_id"
es_modeling["Project Attributes"].time_index
>>> "start_date"
es_modeling["Project Attributes"]['location']
>>> <Variable: location (dtype = categorical, count = 1400)>
es_modeling.get_relationship("Metrics",
                             "Project Attributes")
>>> <Relationship: Project Attributes.project_id ->
Metrics.project_id>
```

Figure 3: Attributes of the modeling *Entityset* containing data from the training period. These attributes highlight the key functionalities provided by the *Entityset* abstraction. Each *Entity* contains an index that uniquely describes each row, and a time index that specifies when each row was first recorded. Columns (which are known as Variables to the *Entityset*) can be easily accessed, and are annotated with semantic type information: the "location" column is a *categorical* variable. Lastly, *EntitySets* know how each *Entity* is related to each other *Entity*, facilitating joins between them for feature calculation. We show the relationship between the "Project Attributes" *Entity* and the "Metrics" *Entity*.

– **Measures** such as the number of defects found or the number of maintenance incidents recorded in the last week or month.

– **Metrics** such as the *cyclomatic complexity* (indicates the complexity of a piece of software), or the *percent non-productive time* in the last week of month.

– **Subjective assessments** such as the financial status in the current reporting period, or comments on the current level of risk.

4.1 *Entityset* Representation

Entityset is a relational representation of data that consists of Pandas DataFrames (tables) connected by foreign-key relationships between columns, along with additional metadata. Each table in an *Entityset* is called an *Entity*, which includes the actual data in the form of a Pandas DataFrame, as well as the semantic types of each column, a pointer to the `index` column that uniquely identifies each row, and an optional pointer to the `timeindex` column that specifies the time of occurrence of each row.

This `timeindex` column allows `cutoff_times` to be specified in the automated feature engineering algorithms, which will link rows across tables and only use information from before the `cutoff_times` specified for a particular row in a particular table.

A full description of an *Entityset* can be provided via a `metadata.json` file. This file lists the `index`, `timeindex`, relationships, and semantic variable types for each entity. An *Entityset* can thus be created in memory with a pointer to the location of the relevant data on disk, and a `metadata.json` file.

Future data can be loaded into a new *Entityset*, and the two structures can be easily joined together to build new features and make new predictions.

We formed an *Entityset* from our raw data. Without describing every field, we present a view of how we represent the data in Figure 3. Each project has an entry in the *Project Attributes* table, along with its start date and about 20 static attributes, such as its industry segment and geographic location. Project managers fill out weekly and monthly reports about each project, which are contained in several different tables. Time-Varying Attributes contains regularly recorded *attributes* and subjective *assessments* about each project, *Metrics* contains regularly recorded *metrics* about each project, and *Measures* contains regularly record *measures* about each project. *High Risk List* records lists of projects each week that are deemed to have something risky about them, along with handwritten comments. Finally, *Attention List* records a list of projects each week that should be given special attention by senior management.

5 Prediction Engineering

Converting the high-level problem of *predicting the likelihood of missing a requirement on a metric* into a concrete, supervised learning problem involves a few steps: (1) quantitatively specifying the outcome to predict, (2) specifying key parameters pertaining to how the model will be put to use in real time, and (3) finding the past occurrences a.k.a training examples (Kanter et al., 2016). The first two steps require the *subject matter experts* to provide the system with inputs and parameters, while the third step is automated using a search algorithm (Kanter et al., 2016).

The outcome $f(.)$ to predict: The formal specification of the outcome is broken into several pieces so that the *subject matter experts* can easily change things without having to write software.

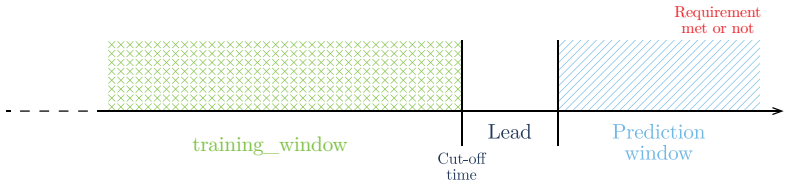

Figure 5: In the weeks corresponding to the prediction window, we evaluate whether all required delivery metrics are met. The lead corresponds to how far ahead we would like to predict this outcome, and the historical window represents the amount of historical data we can use to predict the future outcome. In our study, the historical window was set to entire previous length of the project, the lead to 4 weeks, and the prediction window to 56 days.

– **Metric/value to predict**: For predicting the likelihood of meeting a requirement, two relevant fields in the data were recorded weekly. These were `no_of_reqs_met` and `no_of_reqs`. The *subject matter experts* suggested that the *ratio* of these two fields would indicate what proportion of `requirements` were met, and would be valuable to predict ahead.

– **Over what time period?**: With these two metrics/-fields recorded weekly, our next question was whether we should make predictions for *one* week at a time or somehow aggregate multiple weeks. In prediction engineering, this is called determining the *prediction window*. In our case, the prediction window was set to 56 days (8 weeks).

– **Aggregation:** With the prediction window set at 8 weeks, we now have several values for the two fields from which we aimed to create a ratio. We calculate the *ratio* as:

$$value = \frac{\sum_{pw} no\_of\_reqs\_met}{\sum_{pw} no\_of\_reqs} \tag{1}$$

– **Labeling operation**: The aggregation above gives us a continuously valued number. In our scenario, *subject matter experts* were interested in whether this number was below 1, regardless of what the value was. In other words, they wanted to predict whether even a tiny fraction of *requirements* would not be met. As a result, we converted the value we would like to predict to a binary number by applying:

$$L = \begin{cases} 1 & \text{if } value = 1 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Per **ML 2.0** the outcome function $f(.)$ can be given a `project_id` and a `timestamp` and the *Entityset* and it can calculate the label for that project. This function in our implementation is called `generate_labels_at_cutoff` executes the operations above for a given project.

Parameters for operationalization: We next asked the *subject matter experts* how they imagined this model would be used: How soon after the start of the project would they like to make predictions (`min_training_data`)? How far ahead would they like to predict (`lead`)? These conditions also specify how we extract training examples.

– **Lead time**: This is the amount of time ahead we would try to make a prediction. The duration of the prediction window starts at the end of the lead time. The *subject matter experts* set this at 28 days (4 weeks). This gives them enough time to be able to take action.

– **Minimum training data** We only used projects with a minimum amount of training data (28 days) available before the cutoff time. If the cutoff time was before the first 28 days of a project, we did not include that project in the model.

Search parameters: We decided to choose only one negative training example per project. We searched in the period between July 2012 and July 2014 to find the one training example.

In specifying the "*prediction problem*" we followed the formal procedure prescribed by (Kanter et al., 2016). We designed a search algorithm described in (Kanter et al., 2016) and exposed its parameters through a higher-level function as shown in Figure 6. This enables *subject matter experts* to change all the parameters. The output of this function is `labels_with_cutoffs` which is a three tuple given by $<$ `project_id`, `cutoff_time`, `label` $>$

6 Feature Engineering

Developing a predictive model for a dataset as complex as this one that is accurate enough for industry deployment requires feature engineering — transforming the historical data into high-level statistical patterns that provide a rich description of a project at a particular point in time. The alternative is only using the static attributes of each project, which are highly unlikely to be predictive as the project advances in time, and also produce a model that is irrelevant for an ongoing deployment use case. The predictions would not change over time, and it would be impossible to change

```
labels_with_cutoffs = search_training_examples(es_modeling,
                                               min_training_data='28 days',
                                               lead='28 days',
                                               prediction_window='56 days',
                                               generate_labels_at_cutoff,
                                               iterate_by='7 days')
```

Figure 6: Generating labels with cutoff times from hyperparameters specified for prediction engineering. This function exposes different settings that the *subject matter experts* may want to change. `iterate_by` and `reduce` are internal parameters for the search process.

any of the features used by the model on the fly. Therefore, to provide anything of value to project managers, time-varying features must be used.

6.1 Automated feature generation: Deep feature synthesis

Feature engineering was complicated by the number of fields recorded weekly or monthly per project (351). Manually constructing features would have been tedious, and would have required extensive back-and-forth communication with the *subject matter experts* who understand the data. Many of the fields had names that were hard to parse, and were highly specific. Instead, we performed automated feature engineering to varying degrees of complexity, and incorporated *subject matter expert* advice at a simple level that minimized the need for two-way conversation. Automation was achieved both through the abstractions provided by `Featuretools`, as well as by the `Featuretools` implementation of the *Deep Feature Synthesis* algorithm.

Deep Feature Synthesis: The purpose of Deep Feature Synthesis (DFS)(Kanter & Veeramachaneni, 2015) is to create new features for machine learning using the relational structure of the dataset. The relational structure of the data is exposed to DFS as *entities* and *relationships*.

An entity is a list of instances and a collection of features that describe each one — not unlike a table in a database. The static *project_demographics* entity consists of the project ID, along with the static features that describe each project, such as the geographic region or scope of work.

A relationship describes how instances in two entities can be connected. For example, the time-varying metrics recorded weekly per project in the *Metrics* report can be thought of as another entity, connected to *Project Attributes* by a relationship. Because each project has many recorded metrics over time, the relationship between *Project Attributes* and *Metrics* can be described as a "parent and child" relationship, in which each parent (project) has one or more children (recorded metric).

Given the relational structure, DFS searches its built-in set of *primitive feature functions*, or "`primitives`," for the best ways to synthesize new features. Each "`primitive`" is annotated with the data types it accepts as inputs and the data type it outputs. Using this information, DFS can stack multiple primitives to find *deep features* that have the best predictive accuracy for a given problem. The primitive functions in DFS take two forms.

- Transform primitives: This type of primitive creates a new feature by applying a function to an existing column in a table. For example, the `Weekend` primitive could accept the date column from the 'application_measures' tables as input and output a column indicating whether the metric was recorded on a weekend.
- Aggregation primitives: This type of primitive uses the relations between rows in a table. In this dataset, each project contains many time-varying metrics associated with it. To use the relationship between projects and its metrics, we could apply the `Sum` primitive to calculate the number of times a particular requirement on a metric was missed in the past.

Synthesizing deep features: DFS can apply a second primitive to the output of the first. For example, we might first apply the `Percentile` transform primitive to determine the percentile against all other projects of the recorded metric *total full time employees*. Then we can apply `Std` aggregation primitive to determine standard deviation of the percentile for a project of its number of full time employees over time. If the project contained extreme variability in the number of full time employees compared to other ongoing projects, this could be an indicator that something is wrong with the project and indicative of an impending problem.

Listing of Feature Sets Tested: We leveraged `Featuretools` to build 6 different feature sets. Three of these feature sets were manually defined using the abstractions provided by `Featuretools`, while the rest were either purely generated by the underlying *Deep Feature Synthesis* algorithm or by a combination of *Deep Feature Synthesis* and manual features. Besides the baseline feature set of purely static attributes, all of the feature

```
feature_list = create_features(es_modeling,
                               agg_primitives=[Mean, Mode, Last],
                               trans_primitives=[Percentile],
                               max_depth=2)
feature_matrix = calculate_feature_matrix(es_modeling,
                                          feature_list,
                                          labels_with_cutoffs)
```

Figure 7: Automated feature generation and computation using specific `cutoff_times` per project. The first call to `create_features()` runs *Deep Feature Synthesis* to generate a list of feature abstractions, which are then passed to the second call, `calculate_feature_matrix`, to compute a feature matrix for each `<project_id, cutoff>` tuple in `labels_with_cutoffs`. he returned `feature_matrix` is the matrix with features computed on the data.

sets use historical *Aggregation* primitives on time-varying metrics, and some of them use *Transform* primitives on static attributes or time-varying metrics. Some feature sets take advantage of stacking multiple primitives on top of each other. The details of these 6 feature sets are described in Section 9.

6.2 Computing features from the data

To rapidly test our end-to-end system, we chose the two smallest sets of historical, time-varying features (Feature Set 3 and Feature Set 5). These sets were small enough to compute in under five minutes, and yet still allowed us to fully test the feature engineering and feature selection components of the system. Once we were satisfied with our system and had the ability to reliably test it, we computed the other, more complex feature sets.

We took advantage of the `Featuretools` feature engineering framework to drastically speed up the development cycle here. `Featuretools` helped in the following ways:

Preventing label leakage: Solving a predictive problem using machine learning assumes that the precursors of a certain outcome in the past are similar to those that would happen before that outcome in the future. Therefore, we model the data by selecting points in time in the past for many projects, and using them to predict, at that time, whether a particular project will miss a delivery metric requirement four weeks in the future. Those points in time are called `cutoff_times`. To generate features for each project to use as training examples, we only use the data before each project's `cutoff_time`. Otherwise, information from the time the label was generated can leak into machine learning algorithm. This would cause the trained model to predict more poorly in a live deployment than on any test set using historical data.

To illustrate this point, consider a feature that aggregates the number of times a project has missed its delivery metrics requirements. In a live deployment, computing that feature is straightforward: pull out all the data for a project and count the number of past misses. To train or test a machine learning model, however, we must simulate a particular point in time in the past, and only extract the misses from before that point in time.

Feature
MEAN(percent_req_compliance)
STD(percent_req_compliance)
LAST(no_of_reqs)
MEAN(resolution_time_performance_percent_incident)
MEAN(no_of_reqs)
STD(incident_resolution_time_performance_p3)
STD(resolution_time_performance_percent_incident)
LAST(no_of_reqs_met)
COUNT(Time-Varying Attributes WHERE delivery_status = good)
STD(no_of_reqs_met)

Table 1: Table showing the top 10 most important features (according to the Random Forest) for the feature set with the highest precision. Six of these are built off a project's history of compliance with the required metrics; three pertain to other metrics; one pertains to a subjective assessment.

`Featuretools` provided a built-in way to specify `cutoff_times` for each project when computing features. This allowed us to iterate on different feature sets without manually writing error-prone scripts to cut off the data. [2]

7 Modeling & Operationalization

After prediction engineering and feature engineering, we have data available for modeling in the form of a feature matrix and corresponding set of labels. Each row of the feature matrix and each label is associated with a particular

[2] Note that multiple `cutoff_times` were supplied per project in the test and threshold-tuning sets. When multiple `cutoff_times` are specified for a single project, `Featuretools` computes features multiple times and returns different values for each `cutoff_time`.

project and point in time. Additionally, the data is split in three sections, separated by two points in time: $t1$ = July 2014 and $t2$ = January 2015. The period before $t1$ was used for training the model, from $t1$ to $t2$ for tuning the decision threshold, and after $t2$ for testing the model. There were 1 400 examples (one example per project) in the training set, 11 168 examples (995 projects, one example per project per week) in the threshold-tuning set, and 11 168 examples (1 005 projects, one example per project per week) in the testing set. Each period of data contains a corresponding feature matrix.

Step 1: Training a model: Using the labeled data in the training set, we built machine learning models for each feature set using the following steps:

1. Remove outlier examples whose feature values are all missing or default values.
2. Impute missing values with the most frequent value for that feature among all examples.
3. Scale the values of each feature to have unit mean and standard deviation.
4. If the number of features in the feature set is greater than 250:
   (a) Train a Random Forest with 1 000 estimators.
   (b) Select the top 100 features by importance (feature importances are built into the Random Forest model).
5. Train a Random Forest with 100 estimators (using the new set of 100 features if feature selection was performed).

Step 2: Operationalization of the model: Once the machine learning model was built, we then used labeled data in the threshold-tuning set to determine a decision threshold as follows:

1. Use the Random Forest model to generate real-valued prediction scores between 0 and 1 for every example in the threshold-tuning set.
2. Quantize 1000 possible threshold values at equal increments between 0 and 1, separated by 0.01.
3. For each threshold, binarize predictions by setting a prediction to True (= Will miss a requirement) if greater than threshold.
4. Calculate the true positive rate (recall).
5. Select minimum threshold such that true positive rate is at least 0.9.

Step 3: Validating the model performance: Now, armed with a fitted model and a threshold to binarize its outputs, we made predictions for each example in the test set. We estimated the performance on this data by computing the precision, recall, and false positive rate compared to the actual labels.

Step 4: Reporting performance based on repeated trials: Lastly, in order to build confidence in our final performance estimate, we repeated Steps 1 and 2 30 times, using different random seeds for the number generators within the Random Forest in both the feature selection step and the modeling step. The final performance estimates that we report in Table 2 are averaged across these 30 trials.

Step 5: Prepare deployment system: Each of the 30 trails produce a slightly different decision threshold. For deployment, we select a random model from the 30, and used the mean of the 30 decision thresholds.

7.1 Results from training phase

Comparison to Baselines: The first question to ask when building a predictive model is whether it beats a naive baseline. To that effect, we built several baselines to compare our results against. These included some simple non-machine-learning based predictors, and one simple machine-learning solution using only static attributes. The non-machine-learning baselines looked at each project's prior history of missing delivery metric requirements, prior to the cutoff time.

- **Any Baseline** predicted that a project will miss a requirement if a project had *any* misses in the past.
- **Mode Baseline** looked at each week in a project's history, and predicted a miss if more weeks than not contained a miss.
- **Last Baseline** predicted a miss if the most recent week for a project contained a miss.
- **Static ML Baseline** trained a Random Forest with 100 estimators using only the static attributes of each project from the `project_demographics` table.

As shown in the Table 2, all of our models built using feature engineering significantly beat all of these baselines. We conclude that our machine learning models are at least providing some added value to the problem. [3]

Automated Feature Engineering & Human Guidance We incorporated guidance from *subject matter experts* in several of our feature sets. To accomplish this, we first asked

[3] In future, a more complex baseline that we will examine in the deployment phase of this project is a comparison to the predictions of human project managers.

Feature Set	Precision	Recall	FPR
Static ML Baseline	0.053 ± 0.001	0.881 ± 0.009	0.723 ± 0.015
Any Baseline	0.056	0.989	0.801
Mode Baseline	0.061	0.921	0.681
Last Baseline	0.063	0.964	0.689
SME-Guided Simple Aggregations	0.092 ± 0.001	0.971 ± 0.002	0.442 ± 0.008
SME-Guided Complex Aggregations	0.103 ± 0.003	0.949 ± 0.004	0.383 ± 0.012
Stacking Complex Aggregations	0.110 ± 0.004	0.954 ± 0.005	0.358 ± 0.015
Simple Aggregations	0.115 ± 0.002	0.946 ± 0.003	0.335 ± 0.008
Complex Aggregations	0.125 ± 0.003	0.968 ± 0.004	0.311 ± 0.009

Table 2: A table describing the results of the experiments we performed."Feature set" briefly describes which set of engineered features the experiment used (see Table 4 for more details). Each experiment used a Random Forest with 100 estimators. If the feature set contained more than 250 features, a Random Forest with 1000 estimators was trained first, and the top 100 features according to built-in importance were selected for use. FPR stands for false positive rate. We include the sample mean and 2 times the standard error from the mean on 30 trials. This error term can be interpreted as 95% confidence that the true mean falls within plus or minus this error term from the reported sample mean.

subject matter experts to identify all fields they thought were predictive of the outcome in questions. Then, we chose a feature set we had already built, and removed all features that were not constructed on top of any of the *subject matter expert*-selected fields. Using guidance from *subject matter experts* in this way we dramatically reduced the number of features while still maintaining some predictive power. While neither of these selected feature sets produced the best model, they both produced models that were much quicker to compute than the original sets prior to removal (≤ 5 minutes compared to 1 hour on the combined training and testing set using a modern laptop computer).

Ranking of models Feature Set 4 (Complex Aggregations) produced the best performance across all measured metrics (precision, recall, and false positive rate). It seems that this feature set struck the best balance between including enough complex features to be able to predict nuances of the missed requirements, but not so many that it overwhelmed the Random Forest. Feature Set 6 (Stacking Complex Aggregations), which included features that stacked on top of each other, not only performed worse than Feature Set 4 (Complex Aggregations), but also than the simpler Feature Set 2 (Simple Aggregations), which did not use *Deep Feature Synthesis*. All three of these feature sets performed better than Feature Sets 3 and 5 (SME-Guided Simple Aggregations and SME-Guided Complex Aggregations), built from fields selected by subject matter experts. This shows the value provided by machine learning in unearthing features that humans may not have thought were predictive.

7.2 Rationale behind several choices

We made several key decisions regarding how we trained models and thresholds and tested/validated our system. These allowed us to produce reliable, accurate estimates of how the system would behave when deployed. In this section we explain why we made certain choices, and key insights we gathered along the way.

Time based data splits We opted for hard, time-based splits instead of cross-validation to better simulate the deployment scenario. This is of particular importance for time-based predictive analytics, where the best simulation involves cutting off the data at past moments in time. Time-based simulation is less important in other machine-learning scenarios where underlying distributions are not likely to vary over time, such as classifying objects in images.

Choice of Machine Learning model & procedure The Random Forest model, used for both feature selection and prediction, uses subsampling to learn multiple decision trees from the same data. We used the implementation in scikit-learn. It was selected due to its fast training time, ability to perform well without fiddling with hyperparameters, and built-in feature importances. To establish baseline results and compare feature sets, we mostly stuck with the default parameter settings, with two exceptions.

We increased the number of estimators to 1000 for the feature selection Random Forest, and 100 for the prediction Random Forest. Each estimator uses a different random subsample of features, and we found that we needed to increase the number of estimators to the order of the number of features for the Random Forest to explore enough feature subsets and produce reliable outputs.

Furthermore, due to our highly imbalanced class labels, we chose the `class_weight` $=$ `'balanced'` setting, which instructs the algorithm to automatically adjust the internal weights of samples based on their class label, so that the

weights are inversely proportional to the class frequencies in the input data.

No AutoML We decided not to pursue optimization of the machine learning model itself or search through different modeling methods, known as AutoML in the machine learning community. Due to our experience with the difficulty of ensuring stable output metrics for a given set of choices for validation, we hypothesized that optimizing machine learning methods and hyperparameters would overfit to our train set, as well as produce unreliable estimates that did not match the true deployment performance. As our deployment continues, we will soon have enough data to begin to optimize methods and hyperparameters, while testing the performance on live data.

Choice of metric Since projects are overwhelmingly likely to meet all of their requirements, our model's usefulness increases with how sensitive it is to the early warning signs of future missed requirements. Therefore, we fixed the recall at .9, meaning that if the model predicts a missed requirement on a randomly selected project in the sample set, then there is a 10% probability the model actually has a missed requirement (Goutte & Gaussier, 2005). With this value fixed, we focus on increasing the precision and lowering the false positive rate. In more domain-specific terms, the precision is the probability that a randomly selected project in the sample set actually had a missed requirement given that the model predicted it would (Goutte & Gaussier, 2005). The false positive rate is the number of times the model predicted a missed requirement when one did not exist, divided by the total number of non-misses in the test set.

One caveat of measuring precision is that it is dependent on the ratio of positive to negative examples. We circumvent this problem by selecting as many examples as possible in the testing set, rather than doing any kind of upsampling of one class.

A data split for identifying threshold: One could question why we needed a completely separate set of data to tune the decision threshold. We found that setting the threshold on the same data used for training resulted in severe overfitting, with the end result being a much lower true positive rate (recall) than expected on the test set. Another option one might consider in the modeling phase is to set the threshold on the test set itself. However, this data would not be available in a live deployment, so must be off-limits during modeling to realistically estimate performance.

Different process for extracting training examples: As discussed in Section 5, we extracted a single label per project in the training set, but as many labels as possible in the testing set (one per week per project) and threshold-tuning set. This resulted in 1 400 examples in the training set, but 11 168 examples each in the testing set and threshold-tuning set. Being able to set the threshold and test using many more examples allowed us to produce more reliable performance estimates. Selecting a single example per project in the testing or threshold-tuning sets left us too few positive examples to produce statistically valid results, even with cross-validation.

Repeated trials The Random Forest is a nondeterministic algorithm, so an individual trial of two feature sets could produce metrics with swapped performance relative to each other. To mitigate this potential for error in ranking feature sets against one another, **ML 2.0** suggests conducting repeated random trials of the modeling procedure. Using the Law of Large Numbers and the Central Limit theorem, It's possible to find the mean expected value of the score of a collection of runs to arbitrary precision by increasing the number of runs. We reran the entire modeling procedure, including feature selection, threshold-tuning, and machine learning, 30 times for each feature set. The presented results include the mean and two times the standard error from the mean across all trials. The standard error from the mean is calculated as follows:

$$\sigma_M = \frac{\sigma}{\sqrt{n}}$$

Where σ is the overall standard deviation, and $n = 30$ is the number of trials.

This error term σ_M can be interpreted as 95% confidence that the true mean falls within $\pm\sigma_M$ from the reported sample mean.

On any given trial, it's possible that the ranking of feature sets will vary. However, a consistent and confident ranking emerges after n trials by reducing the noise inherent in the Random Forest. Random Forests, as their name implies, exploit randomness in their internal algorithms. Furthermore, the Random Forest responsible for feature selection may produce a different ordering of features, leading to a different set of 100 features used for modeling. Indeed, the top 100 features were observed to be slightly different for different random seeds.

8 Deploying the predictive system

The biggest enabler of **ML 2.0** is the ability to take machine learning models to production. The automation tool `Featuretools` and the `APIs` that come with it make this possible by providing three items: (a) a standard, well-defined output from the training phase; (b) a standardized set of `APIs` to test, validate and deploy the model; (c) a series of tests and validations.

8.1 Standardized output from training phase

ML 2.0 formally specifies several components that need to be present in a delivery from training to deployment.

```
fl = ft.load_features("fl.p", es_updated)
fm = ft.calculate_feature_matrix(fl,cutoff_time="8/1/2017")
```

Figure 8: `API` **for computing features on the new data**. First, the feature list that specifies the relevant features and *Entityset* is loaded from disk. Then, the feature list is passed to the feature computation logic along with a particular cutoff time. Note: this snippet assumes the *Entityset* is already loaded in memory as `es_updated`, and ft refers to the `Featuretools` package.

```
fl = ft.load_features("fl.p", es_updated)
fm = ft.calculate_feature_matrix(fl, cutoff_time="8/1/2017")
labels = generate_labels_at_cutoff(es_updated, cutoff_time="8/1/2017")
model = load_model("model.p")
predictions = model.predict(fm)
```

Figure 9: `API` **to roll back in time and test**. After the steps laid out in the previous diagram, labels are generated for each project in the updated *Entityset* at the specified cutoff time. The model is loaded from disk, and passed as input for the new feature matrix to generate predictions for these projects at the same cutoff time. The deployer can compare the true labels to the predictions, and generate performance metrics to evaluate the model. Note: this snippet assumes the *Entityset* is already loaded in memory as `es_updated`, and ft refers to the `Featuretools` package.

```
fl = ft.load_features("fl.p", es_updated)
fm = ft.calculate_feature_matrix(fl, cutoff_time=datetime.now())
model = load_model("model.p")
predictions = model.predict(fm)
```

Figure 10: `API` **to deploy on live data**. The first several steps proceed as laid out in the previous diagrams. In several weeks (after the *lead* time), the deployer can compare the true labels to these generated predictions. Note: this snippet assumes the *Entityset* is already loaded in memory as `es_updated`, and ft refers to the `Featuretools` package.

This output goes beyond a *learned* model. It also includes meta-information about the data transformations that need to occur, and about data properties that must remain the same in deployment. Equally importantly, it provides software to transform new raw data into a machine learning input that uses the same *function* calls to the same library used for training. Below, we describe these components of the delivery.

Model provenance: A JSON file, `model_provenance.json`, containing an estimation of the expected performance, a decision threshold setting for the classifier's predictions, and a list of data fields and types that must be present in the new data in order for features to be successfully computed. Optionally, statistics about each of the features – for example, min, max and others – can be passed as well.

Model and features: A feature list file that specifies all of the computations necessary to construct a feature matrix, containing the same features as those the model was trained on, given an *Entityset*. For deployment, the *Entityset* provided would contain both the data from the training period and from the deployment period. A file containing a serialized, fitted model to be used to make predictions.

Software: A packaged Python script that contains a few utility functions and installs necessary dependencies, such as `Featuretools`, loads the feature list and the model, add new data to entity set, computes features on new data, and makes predictions.

8.2 Standardized set of `APIs`

The same software and `APIs` that are used by the training team are also used by the deployment team. The following `APIs` allow users to integrate new data, make predictions on new data, and test and validate on older data.

Add new data: We used the *Entityset* abstraction provided by `Featuretools`. It has an `add_new_data()` method that provides a built-in way to merge multiple *EntitySets* together and thus incorporate new data.

Compute features at a certain time point: With the updated *Entityset* that now contains the new data, the deployment team can simply load the feature list and use the same `API` to compute features at a certain `cutoff_time`, as shown in Figure 9.

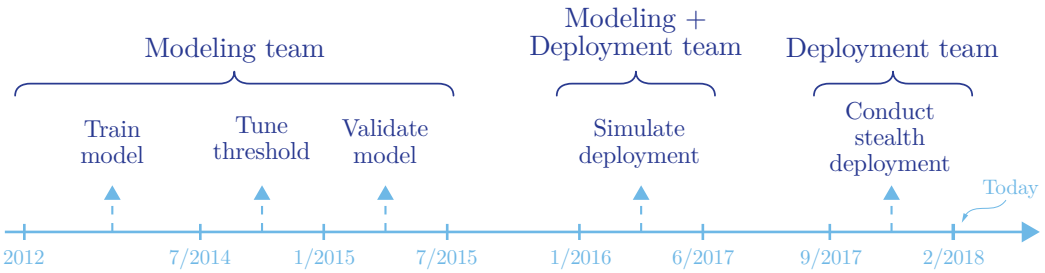

Figure 11: Data used during different phases of model development. The first three periods of time were used exclusively by the modeling team in order to build and validate a model. This included data from 2012 - 7/2014 for training, from 7/2014 - 1/2015 for tuning the decision threshold, and from 1/2015 to 7/2015 was used for testing/validation. Data in the next period was available to both teams but only used to validate results in production, as a simulated deployment. This period covered data from 1/2016 - 6/2017.The final deployment was done on the data from 9/2017 to 2/2018. The model is in use at the time of the writing of this paper.

Roll back and test/validate: During deployment, teams will inevitably want to test the predictive system at different past points in time. This allows them to test the software, validating the model by checking its predictions at different points in the past and comparing them with the actual labels. [4] Validating the model against data from the past creates confidence, familiarity and trust in the predictive system. To enable this, by specifying a `cutoff_time` point in `Featuretools` and an *Entityset*, `Featuretools` automatically filters out data past this time point, and allows users to generate labels, compute features, make predictions using the model, and evaluate for that point in time. 9 shows this particular `API`.

8.3 Testing, validation and deployment

Figure 11 presents the timeline for the data presented to the system up until the point in time at which this paper was written (February 2018).

Training: The first data portion was the largest, consisting of data from early 2012 to July 2014. This was used to train a machine learning model. We used the second portion, consisting of data from July 2014 to January 2015, to tune a decision threshold, and a portion from January 2015 to July 2015 to validate our model. The training methodology and results are presented in Section 5. The validation done within the training phase, however comprehensive, is always revisited during deployment using data never used before. Next, we describe the testing and validation performed in deployment.

Stage 1: Correctness & usability testing The first test deployment teams must perform is whether they can reliably use the provided system. To achieve this, our deployment team gathered data from August 2016 to September 2017 that was not available to the training phase, successfully augmenting the dataset in the *Entityset* using `add_new_data`. With this system, they can compute a feature matrix using data before a particular `cutoff_time`, and generate predictions using the model.

Stage 2: Validating for accuracy: At this stage, the predictive system is used to generate predictions on data for which one knows the ground truth. It allows the deployment team to measure whether the model can produce accurate predictions on data not provided to the training phase. To do this, we used the data from February 2016 to September 2017 and the `roll back and test` process we described above. We selected multiple `cutoff_times` during the period, made predictions, and compared these predictions against the actual outcomes. This validation reveals any unknown biases and mismatches/inconsistencies in the data, inconsistencies in the understanding of what data should be given to the model and even data drifting. Ultimately, it establishes the trust and confidence in the predictive system that deployment teams require in order to take it live. Through multiple rounds of validation, we identified and solved a number of these issues. These are summarized in Table 6. We are currently processing validation results from this 53 week period over data (February 2016 to February 2017) which has never been used to model. We will update this document with those results as they become available (as an appendix).

Stage 3: Stealth deployment on live data: Finally, after testing for reliability and accuracy, we were ready to deploy on the live data. Since September 2017, we have created predictions every week for numerous projects. This phase is called *stealth*, since the predictions are not directly delivered to the project managers, but are evaluated by the subject matter experts.

[4] As we will show in the next section, this particular past data is usually not part of the training data.

```
es_updated = add_new_data(es_modeling,
                          new_data_path,
                          metadata_json_path,
                          model_json_path)
```

Figure 12: `API` **for updating the *Entityset* with the new data**: The old *Entityset* `es` is transferred, along with the path to where the new data resides, the `metadata.json` file providing the full specification of the *Entityset*, and the `model_provenance.json` file with information about each of the data fields used to build feature matrices. The `add_new_data` function checks that all fields used for feature computation are present and typologically consistent in the new data, appends each data table with new data, and makes sure indexes (including time indexes) are properly sorted. The result is a unified view of the full dataset, allowing access to both old and new data.

Fields	351
Updates	40389171
Years	6
Training	
Projects	1400
Samples	1400
Threshold-Setting	
Projects	945
Samples	11168
Testing	
Projects	1005
Samples	11168
Deployment	
Projects	1493

Table 3: Statistics about the data for both modeling and deployment.

9 Final Remarks

Using the **ML 2.0** paradigm to remain deployment-focused and to maintain a close relationship between the modeling and deployment teams at every step of the process, we were able to build a productionized machine-learning system to augment project managers in a way that has never been done before. Our contributions can be summarized in the following ways:

- We not only deployed a well-tested model, but we also incorporated it into a UI that *SMEs* could interact with, enabling a complete, usable system.
- Automation tools took the burden off of the modeling team from implementing a myriad of custom, error-prone computations, allowing the team to focus on higher-level problems like correct validation and choosing the right feature sets.
- The concept of `cutoff_time` was critical at every stage of the project, from prediction engineering, through feature computation, through machine learning & validation, and finally deployment validation.
- Explicit organization, such as the use of **ML 2.0** concepts like *EntitySets*, the `metadata.json` file", and the `model_provenance.json` file, significantly improved our confidence and reduced the time to first deployment.

Finally, the key takeaways from this project are not restricted to the efficacy of machine learning on predicting whether delivery metrics will be met. Rather, we have demonstrated a complete system that works in the real world, on continually updating live data. Furthermore, we have provided a concrete example of an **ML 2.0** project whose steps can be applied to any time-based machine learning project. All `API`s and abstractions referenced, function calls made, validation methods used, and problems faced are transferable to an extremely diverse set of real-world problems for which deployable machine-learning models are possible. We hope that our process will be used by projects in completely different industries to actually deploy machine learning solutions, and welcome anyone to use any of the `API`s and methods we implemented and discussed.

References

Goutte, Cyril and Gaussier, Eric. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In Losada, David E. and Fernández-Luna, Juan M. (eds.), *Advances in Information Retrieval*, pp. 345–359, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31865-1.

Hu, Yong, Huang, Jiaxing, Chen, Juhua, Liu, Mei, and Xie, Kang. Software project risk management modeling with neural network and support vector machine approaches. In *Natural Computation, 2007. ICNC 2007. Third International Conference on*, volume 3, pp. 358–362. IEEE, 2007.

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

Kanter, James Max, Gillespie, Owen, and Veeramachaneni, Kalyan. Label, segment, featurize: a cross domain framework for prediction engineering. In *Data Science and Advanced Analytics (DSAA), 2016 IEEE International Conference on*, pp. 430–439. IEEE, 2016.

Ling, Florence Yean Yng and Liu, Min. Using neural network to predict performance of design-build projects in singapore. *Building and Environment*, 39(10):1263–1274, 2004.

Feature sets

Table 4: Feature Sets.

No.	Name	Description
1	Static ML baseline	This feature set used only the categorical, numeric and Boolean fields from the *Project Attributes* table, corresponding to 22 fields. The 15 categorical fields were One-Hot-Encoded to produce 154 total machine-learning ready features.
2	Simple aggregations	This feature set used all of the static features, and additionally incorporated aggregations of metrics in the *Metrics*, *Measures*, and Time-Varying Attributes tables. The aggregation primitives were limited to the mean (value of each metric), mode (most common value of each metric), and last value (most recent value of each metric). Categorical features were One-Hot-Encoded, and features with all the same value or with all missing values were removed. This resulted in 1 982 features.
3	SME-Guided simple aggregations	This feature set removed all features from Feature Set 2 that did not build on top of fields deemed to have predictive power by the *subject matter experts*. It resulted in 93 features.
4	Complex aggregations	This feature set used all of the features in Feature Set 2. It incorporated five more *Aggregation* primitives, and applied them to every possible column in all four of the time-varying tables in the dataset, as well as six *Transform* primitives applied to the static table. This resulted in 2 706 features. The primitives used are presented in Table 5.
5	SME-Guided complex aggregations	This feature set removed all features from Feature Set 4 that did not build on top of fields deemed to have predictive power by the *subject matter experts*. It resulted in 201 features.
6	Stacking complex aggregations	This feature set took advantage of the ability to stack primitives on top of each other in *Deep Feature Synthesis*. It used all of the primitives in Feature Set 4, and then combined the aggregation primitives with the *Transform* primitives to produce features that, for instance, calculated the rank among all projects of the mean of a project's historical percentage of met delivery metrics. It resulted in 6 899 features.

Table 5: Primitives.

Name	Type of Primitive	Description
Last	Aggregation	Last value in the training window
Mean	Aggregation	Mean of values in the training window
Mode	Aggregation	Most common value in the training window
Sum	Aggregation	Sum of values in the training window
Std	Aggregation	Standard deviation of values in the training window
Trend	Aggregation	Slope of linear trend line fitted against time to the values in the training window
Count	Aggregation	Count of values in the training window
Skew	Aggregation	Statistical skew measure of the distribution of values in the training window
Percentile	Transform	Percentile of a value compared to all other values in the data
Weekend	Transform	Boolean primitive that is true if underlying date value falls on a weekend
Weekday	Transform	Boolean primitive that is true if underlying date value falls on a weekday
Month	Transform	Month of the year (1-12) of the underlying date value
Year	Transform	Year of the underlying date value
Day	Transform	Day of the week (1-7) of the underlying date value

Table 6: Data quality issues we faced in deployment. These are also common problems to watch out for in any deployed machine learning project. Due to our ability to roll back and test in deployment and information we stored in `metadata.json` and `model_provenance.json` we were able to identify these issues quickly and resolve them.

No.	Issue	Description
1	Amount of data	Making predictions on projects that had just recently started produced poorer results than on those that had existed for more than a few months. In general, we noticed that the more data was available from a given project, the better our model performed on it.
2	Gaps in data	The final dataset contained gaps of several months where data was missing for many projects. This had a detrimental impact on performance in the deployment period.
3	Missing data fields	Several of the data fields were missing in the deployment data. Many of these were able to be added in later, but some had stopped being recorded by project managers and so had effectively disappeared. This is one reason for the reduction in performance on the deployment data.
4	Re-scaled data values	At some point between when the data that we trained on was created and when the data we deployed on was created, the data collection team updated how several fields were computed. In particular, several metrics representing a "percentage" were changed from a ratio between 0 and 1 to a percentage point between 0 and 100. These values needed to be rescaled before using them as input to our machine-learning model.
5	Too few positive examples per week	We computed precision, recall, and false-positive rates every week in the deployment phase. However, in many cases there were fewer than 5 projects with missed requirements each week. Therefore, slight variations week to week would drastically change the computed recall metric. To mitigate this, we needed to average over many weeks for each reported result.