MakeML: Automated Machine Learning from Data to Predictions

by

Isabella M. Tromba

B.S., Mathematics and Computer Science and Engineering,
Massachusetts Institute of Technology (2014)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author . . . .
Department of Electrical Engineering and Computer Science
June 8, 2018

Certified by . . . .
Sam Madden
Professor
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

MakeML: Automated Machine Learning from Data to Predictions

by

Isabella M. Tromba

Submitted to the Department of Electrical Engineering and Computer Science
on June 8, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

MakeML is a software system that enables knowledge workers with no programming experience to easily and quickly create machine learning models that have competitive performance with models hand-built by trained data scientists. MakeML consists of a web-based application similar to a spreadsheet in which users select features and choose a target column to predict. MakeML then automates the process of feature engineering, model selection, training, and hyperparameter optimization. After training, the user can evaluate the performance of the model and can make predictions on new data using the web interface. We show that a model generated automatically using MakeML is able to achieve accuracy better than 90% of submissions for the Titanic problem on the public data science platform Kaggle.

Thesis Supervisor: Sam Madden
Title: Professor

Acknowledgments

Thank you to my parents, Inga and Tony, and my brother Max for their support in all of my endeavors. Thank you to my sister Lara Tromba whose background in data visualization and analytics tools helped in guiding improvements to the user experience. I cannot thank my advisor Sam Madden enough for his support. His involvement was instrumental in turning this project into a reality.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The information technology revolution has brought an exponential explosion in the amount of data collected by corporations, non-profits, and governments. According to recent research by IBM, 90% of the data in existence was created in just the last 2 years, and this has been true for the last 30 years [11]. Lying within these large datasets are keys to advances in medicine, energy, and many other fields. However, there is a shortage of data scientists with the necessary skills in mathematics and computer science to make sense of the data and unlock its potential. According to a recent report from LinkedIn, demand for "Machine Learning Engineers" and "Data Scientists" has grown by a factor of ten over the last five years, faster than any other field [35]. However, there are many people with skills using spreadsheets and other GUI based tools for processing data. If these people could be equipped with the ability to train and deploy machine learning models, many tasks now reserved for professional data scientists could be open to lower skilled workers.

The typical workflow for a data scientist is to begin with features computed from a dataset, a basic model, and a hyperparameter configuration chosen from the data scientist's intuition. The data scientist will then train the model and gather performance metrics. Using the performance metrics, they determine how to modify the learning rate, regularization, or feature preprocessing to improve model performance.

This iterative process is very tedious and oftentimes imprecise. It is our claim that a large portion of this process can be automated.

1.2 MakeML

To make progress towards this goal, we introduce MakeML, a software system that enables knowledge workers with no programming experience to easily and quickly create machine learning models that have competitive performance with models hand-built by trained data scientists. MakeML consists of a web-based application in which users explore datasets, select features, choose target columns to predict, and train models. MakeML then produces a trained model through the following process:

1. It automatically transforms human-readable features into features appropriate for machine learning algorithms through feature encoding and normalization.

2. It automatically fills in missing values, bucketizes numeric features, and creates polynomial feature crosses.

3. It automatically explores a wide range of machine learning models including logistic and linear regressions, deep neural networks, and gradient boosted decision trees.

4. It searches over a large space of hyperparameter configurations including learning rate, deep model architecture, and L_1 and L_2 regularization.

In the rest of this thesis we describe the MakeML user interface, cover related work, discuss the architecture of MakeML, go into detail on the implementation of particular components, benchmark MakeML's performance, and discuss the results of a user study. We conclude with the shortcomings of MakeML and an outline of future work.

Chapter 2

User Interface

MakeML's user interface is designed to make it intuitive for data-literate knowledge workers to configure machine learning jobs. First, the analyst selects a dataset, chooses features using a spreadsheet-like interface, and chooses a target column to predict. When they click "Run Job", they are brought to the training view. The training view displays the current stage of processing that the job is in and the elapsed time. Once training is complete, the results view is presented which displays the parameters of the best performing model, high level performance metrics on the test set, and information about which features were most important to the model. When the user is satisfied with the model they have created, they can use it to make predictions on new data or download it for use elsewhere.

In this chapter, we walk through an example of configuring and training a new model using MakeML.

2.1 Training a New Model

Upon opening MakeML, we can view any past models we have trained in the left sidebar and create a new one by clicking the "Create New Job" button. We now have the option to configure our features using SQL in the *Query Interface*, shown in Figure 2-1 or using the spreadsheet-like *Builder Interface*, shown in Figure 2-2. We can toggle between the two interfaces at any time.

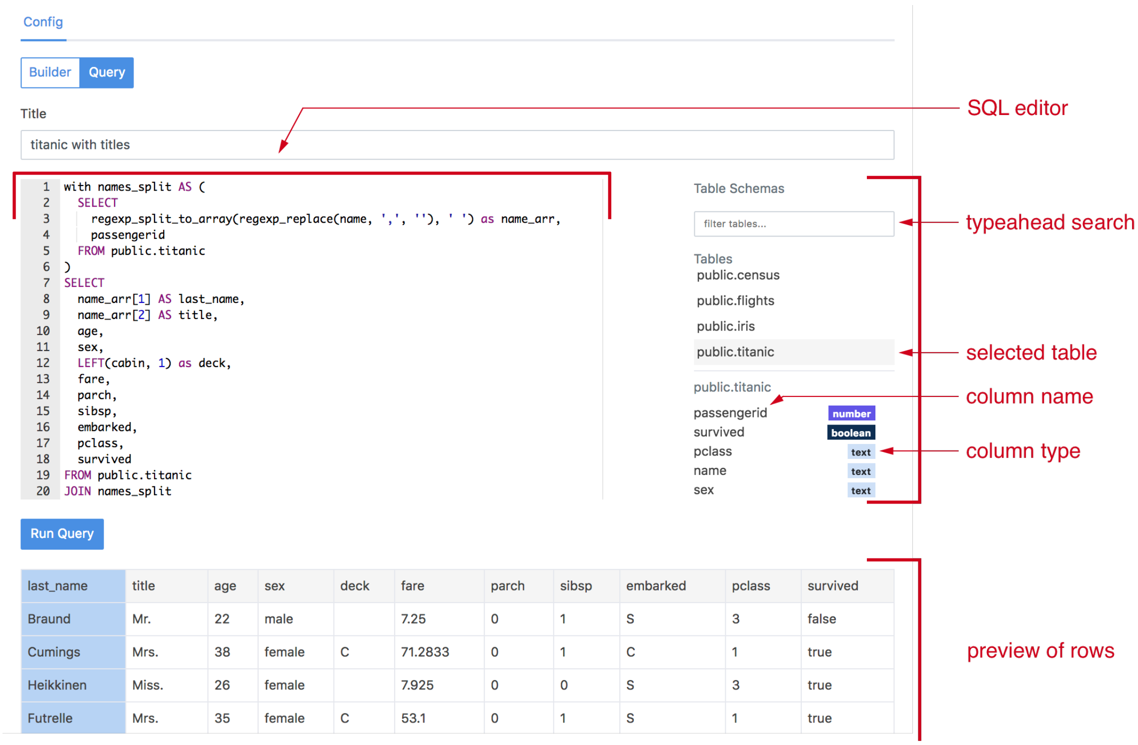

Figure 2-1: MakeML *Query Interface* for feature creation with schema viewer and preview of rows and target column highlighted.

We first select a table to work with. In this example, we choose the "titanic" table which contains the data from the "Titanic: Machine Learning from Disaster" competition from Kaggle [25]. The task is to determine which passengers survived the Titanic disaster. Each row represents a passenger and includes the following features: `survival`, `pclass` (ticket class), `sex`, `age`, `sibsp` (number of siblings/spouses aboard), `parch` (number of parents/children aboard), `ticket`, `fare`, `cabin`, and `embarked` (port of embarkation). For a full description of the dataset, see [25].

2.1.1 Builder Interface

The *Builder Interface* enables users who do not know SQL to easily select columns and compute simple aggregation features from the underlying dataset. After selecting the "titanic" table, we see that all columns are included by default by observing that all of the checkboxes are checked. In addition to the base features, we add two aggregate features. First, we click the "Add Aggregate Feature" button. Then in the "Group

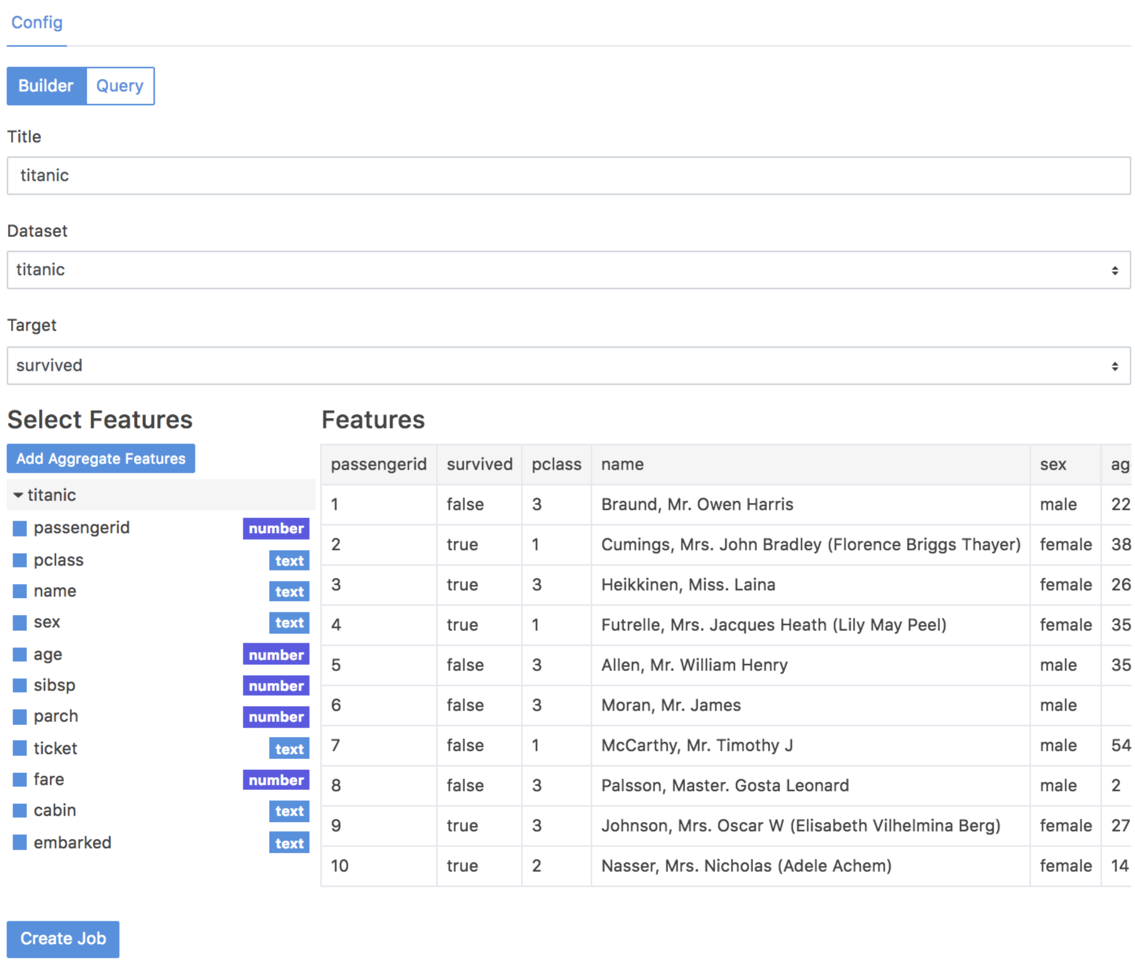

Figure 2-2: MakeML *Builder Interface* for feature creation with preview of rows.

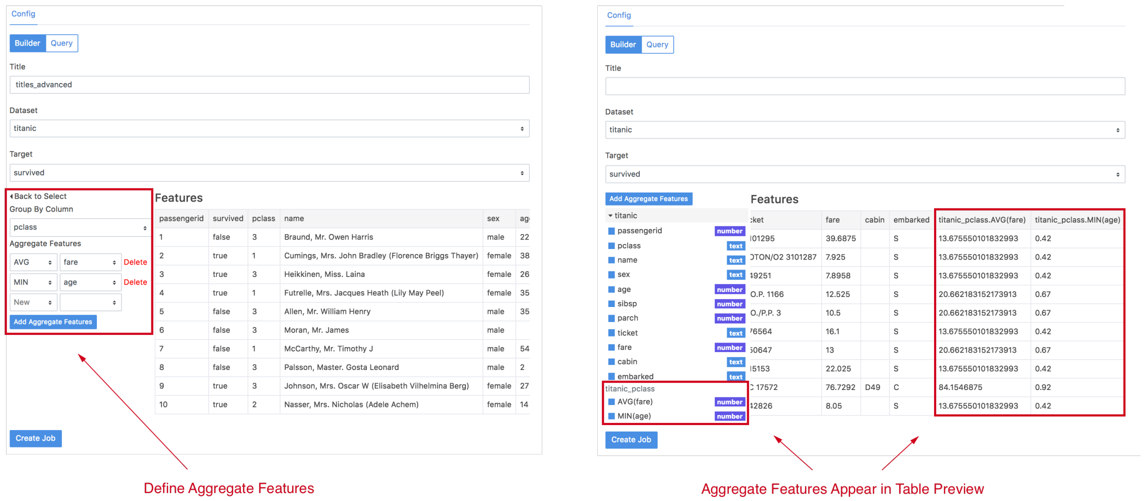

Figure 2-3: Aggregate feature creation in the *Builder Interface*.

By Column" popup we select "pclass" to group by the class of ticket the passenger purchased. Then we add two features, "AVG" of "fare" and "MIN" of "age". This adds, for each row, two new columns, one that reports the average fare and one that reports the minimum age of passengers for the class that passenger was in. Although not applicable to the titanic dataset, for tables that have foreign key relationships, MakeML displays all related tables underneath the main table in the *Column Picker* view. See Figure 2-4.

2.1.2 Query Interface

The *Query Interface* allows us to use a SQL query to define the data to train the model on. For more technically advanced users this gives a high degree of control over the data selected, allowing complex aggregations, joins, and functions on columns. To aid in query composition, the *Query Interface* provides a schema viewer that shows the fields in the selected table along with their associated types. The types are inferred from the underlying relational database schema. For an example of a more complicated query where we use a PostgreSQL regular expression function to extract the `last name` and `title` from the `name` feature, see Figure 2-1.

Figure 2-4: Primary table and related tables in the *Column Picker* view.

2.2 Training

Next, we click the "Create Job" button to begin training. The training interface shows both the currently running stage and the elapsed time since the stage began, as shown in Figure 2-5. MakeML has two modes of training, the "exhaustive" run which takes several hours and the "quick" run which was designed to take no more than 2 minutes on any one of the test datasets.

2.3 Evaluating Model Performance

After the model has completed training, the results view shows the performance metrics of the best performing model along with its hyperparameter configuration and feature weights, if applicable. See Figure 2-7. It also displays the predictions for the test dataset. See Figure 2-6.

After inspecting the hyperparameters, predictions, and performance metrics, we can decide either to make predictions on new data or to download the model for later use.

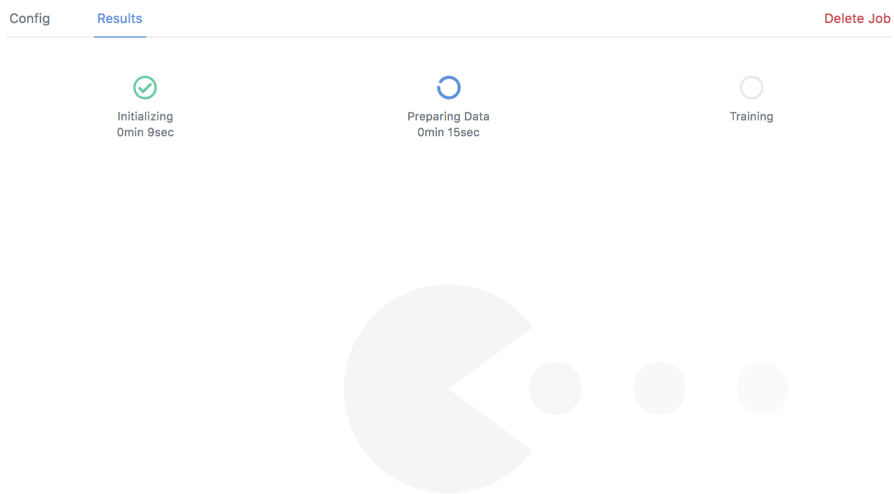

Figure 2-5: Pipeline Stage Status Updates

To make the model performance as easy to understand as possible for less mathematically sophisticated users, we present only three top level metrics: "Accuracy", "Precision", and "Recall". Below are the mathematical definitions of the three metrics as well as the user-friendly help text shown to explain to users the meaning of each metric.

Accuracy

$$\text{Accuracy} = \frac{\text{Number of Correctly Classified Examples}}{\text{Total Number of Examples}}$$

Accuracy: The percent of examples that were correctly classified.

Precision

$$\text{Precision} = \frac{\text{True Positives}}{(\text{True Positives} + \text{False Positives})}$$

Precision: The ability of your classifier to not label negative examples as positive.

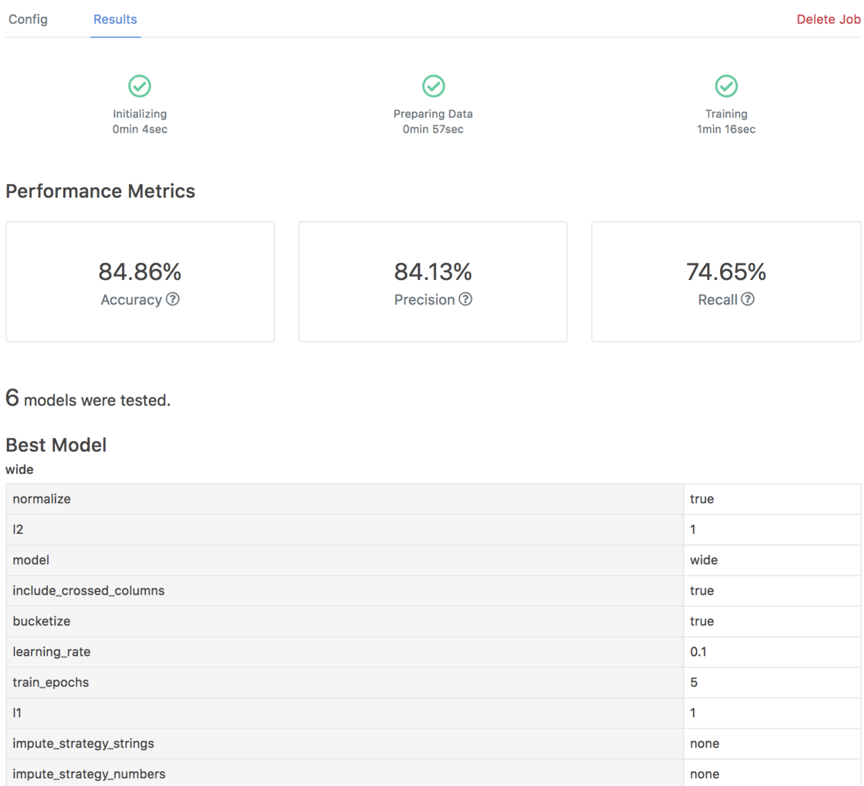

Figure 2-6: Results view showing the number of models trained, the time for each stage in the pipeline, and the hyperparameters of the best performing model.

Recall

$$\text{Recall} = \frac{\text{True Positives}}{(\text{True Positives} + \text{False Negatives})}$$

Recall: The ability of your classifier to find all positive examples.

In all decisions regarding the user interface, we were guided by the goal of making the experience as intuitive and easy as possible. Because users are familiar with spreadsheets, we tried to model the *Builder Interface* using a spreadsheet-like workflow. Simplifying the user interface meant removing metrics that were slightly more complex to explain, like *F1 score* and *Area Under the Receiver Operating Characteristic Curve*. It also meant hiding complex configuration like the `test` and `train` split percentages, column-level configuration of the missing value imputation strategy,

fare fare	0.014134379103779793
parch parch	0.010410487651824951
age_bucketized [8.462, 16.503999999999998)	0.008183369413018227
fare_bucketized [153.69876, 204.93168)	0.007323198951780796
fare_bucketized [102.46584, 153.69876)	0.00638284208253026
age_bucketized [16.504, 24.546)	0.005209038965404034
fare_bucketized_X_sex_indicator fare_bucketized_X_sex_indicator	0
fare_bucketized [409.86336, 461.09628)	0
age_bucketized_X_embarked_indicator age_bucketized_X_embarked_indicator	0
age_bucketized_X_ticket_indicator age_bucketized_X_ticket_indicator	0
age_bucketized [0.42, 8.462)	0
fare_bucketized [256.1646, 307.39752)	-0.001616726047359407
sibsp sibsp	-0.0029339136090129614
age_bucketized [64.756, 72.798)	-0.0030320375226438046
passengerid_bucketized [90.2, 179.4)	-0.00394668523222208
age_bucketized [56.714, 64.756)	-0.00700022280216217
age_bucketized [48.672, 56.714)	-0.00837617926299572
age age	-0.009224136359989643
age_bucketized [32.588, 40.63)	-0.011418240144848824
fare_bucketized [51.23292, 102.46584)	-0.015826376155018806
sex_indicator sex_indicator	-0.0162399560213089
pclass pclass	-0.017068326473236084

Figure 2-7: Feature weights for a trained Titanic model with some features removed for brevity.

among others. While these configurations could be helpful for those with more familiarity in machine learning, these features complicated and cluttered the UI and are not in line with MakeML's goal of creating the simplest and most intuitive interface for configuring machine learning models.

Chapter 3

Architecture

The MakeML machine learning pipeline consists of the stages shown in Figure 3-1 and described in the sections below. MakeML trains many different models, each with differing configurations, and then evaluates the performance of all of these models in order to select the best model.

3.1 Jobs

When the user creates a new job using the web interface, a job definition is created including the SQL query to produce the base features, the target column, and associated metadata. See Figure 3-2 for an example:

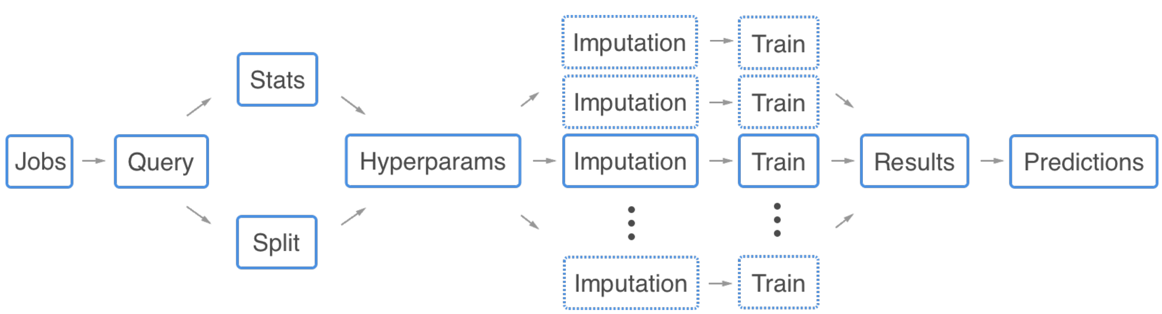

Figure 3-1: MakeML Pipeline Stages.

```
{
    "id":"benchmark_titanic",
    "name":"benchmark_titanic",
    "query":"SELECT * FROM public.titanic",
    "target":"survived",
    "type":"classification",
    "user_id":"966adf53-3bb3-4d6e-9c16-0d85f0972ec3",
    "benchmark":"titanic"
}
```

Figure 3-2: Example job configuration file.

3.2 Query

The creation of the job configuration file triggers the **Query** stage, which is responsible for issuing the query specified in the job configuration file. It writes the result of the query to a CSV file.

3.3 Stats

The **Stats** stage reads in the CSV of training data and computes summary statistics for each column. These will be used in the **Train** stage in order to determine the cardinality of the label vocabulary and to determine the vocabulary for categorical encodings. Stats include the `name` of the column, the count of `null` values, the number of `distinct` values, the `type`, one of `number`, `text` or `unknown`, and for columns with less than 100 distinct values, the array of `values` and a `histogram` of these values. Numeric fields include the `max`, `min`, `mean`, `std`, and `median`. See Figure 3-3 for an example.

3.4 Split

The **Split** stage splits the data into training and testing sets using an 80%/20% split.

```
[
    {
        "null": 0,
        "distinct": 2,
        "values": ["male", "female"],
        "histogram": {"female": 314, "male": 577},
        "type": "text",
        "name": "sex"
    },
    {
        "null": 687,
        "distinct": 9,
        "values": ["C", "E", "G", "D", "A", "B", "F", "T"],
        "histogram": {
            "A": 15,
            "B": 47,
            "C": 59,
            "D": 33,
            "E": 32,
            "F": 13,
            "G": 4,
            "T": 1
        },
        "type": "text",
        "name": "deck"
    },
    {
        "null": 0,
        "distinct": 248,
        "values": [],
        "histogram": [],
        "max": 512.3292,
        "min": 0.0,
        "mean": 32.204207968574636,
        "std": 49.6934285971809,
        "median": 14.4542,
        "type": "number",
        "name": "fare"
    },
    ...
    {
        "null": 0,
        "distinct": 2,
        "values": ["f", "t"],
        "histogram": {"f": 549, "t": 342},
        "type": "text",
        "name": "survived"
    }
]
```

Figure 3-3: A subset of column statistics output from the **Stats** Job.

```
hyperparams_tf_deep = dict(
    model = ['wide-deep', 'deep'],
    learning_rate = [.001, .01, .1],
    l1 = [0.1, 1],
    l2 = [0.1, 1],
    include_crossed_columns = [True],
    bucketize = [True],
    normalize = [True],
    train_epochs=[5, 20],
)
```

Figure 3-4: An Example Hyperparameter Grid for Deep Models.

3.5 Hyperparams

The **Hyperparams** stage configures the grid of hyperparameters for the train stage. A significant amount of research has gone into optimal algorithms for hyperparameter tuning, as covered in the related work section. The current implementation of the hyperparameter and training stages is a grid search across a large number of hyperparameter configurations. Naively, there are currently two grids defined, one for quick training, ≈ 100 models, and one for testing a large number of model configurations, ≈ 1200 models.

3.5.1 Cross-Validation

We use k-fold cross validation in order to determine the optimal hyperparameter configuration. When comparing the performance of hyperparameter configurations on the same test set, there is a risk of overfitting to that test set. One strategy employed to tackle this problem is to split the training data into three separate sets, `train`, `validate`, and `test`. The hyperparameter configurations are tested on the validation set, the best hyperparameter configuration for the validation set is chosen, and then the final evaluation happens on the test set. The problem with this approach is that it significantly limits the amount of data that can be used for training. Cross-validation solves this problem by splitting the training set into k different subsets. The model is trained on $k-1$ of the subsets and then evaluated on

```python
param_grid = dict(
    n_estimators=[100, 200, 500],
    learning_rate=[.01, .1],
    max_depth=[4, 6]
)

kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
xgb_model = xgb.XGBClassifier()
clf = GridSearchCV(xgb_model, param_grid, cv=kfold)
clf.fit(X_train, y_train, verbose=0)
```

Figure 3-5: K-Fold Cross-Validation with parameter grid defined for XGBoost model.

the final subset. The performance of the model and hyperparameter combination is the average performance across all k runs [29][19].

In this way, we do not waste data by holding it in a separate validation set and we are able to address the issue of "leaking" information into the model which causes the models' performance metrics to be overly optimistic with regards to generalization [32].

3.6 Imputation

Some machine learning algorithms are able to handle missing values. Other algorithms, however, require all values to be non-null [14]. The **Imputation** stage reads in the result of the query and fills missing values in each column if necessary. For example, in the Titanic dataset, 687 out of the 891 total passengers have a missing value for the `cabin` field. One option is to drop all rows containing missing values, but this would reduce the size of our training set significantly. As an alternative, we can fill in these missing fields with reasonable values. For string or categorical data, often times we use the most frequently occurring value [14]. Using the mode strategy for the `deck` column, we would fill all missing values with the `deck` C. Another more sophisticated approach is to impute the missing values using another machine learning algorithm. MakeML uses KNN and mode imputation for categorical columns and KNN, mode, and mean imputation for numeric columns. The KNN algorithm has hyperparame-

deck	count
	687
A	15
B	47
C	59
D	33
E	32
F	13
G	4
T	1

Table 3.1: Deck and Counts for Passengers on Titanic

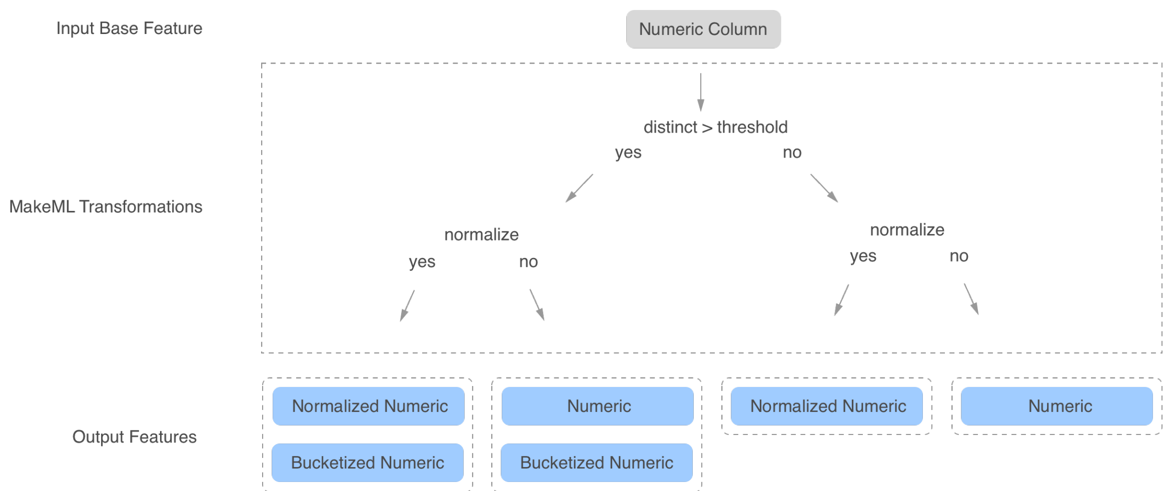

Figure 3-6: MakeML numeric base feature transformations.

ters of its own that could be tuned and included in our grid search, however, for the sake of simplicity, MakeML chooses reasonable default hyperparameters.

3.7 Train

The training stage is the longest running and most important stage in the pipeline. One job is created in the Train stage for each hyperparameter configuration. The **Train** stage performs both the feature engineering and trains the model. The transformations that MakeML performs on each base column depends on the type of the column. For the transformations MakeML does on numeric columns, see Figure 3-6 and for the transformations that MakeML does on string columns, see Figure 3-7.

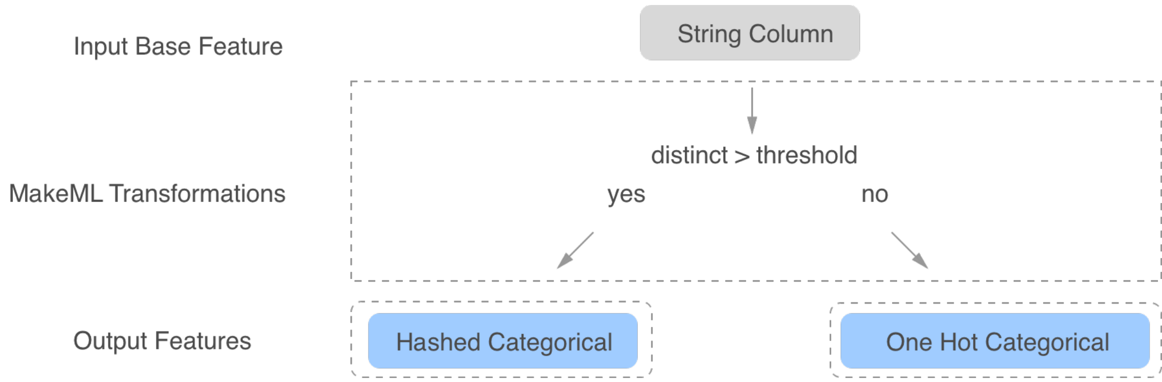

Figure 3-7: MakeML string base feature transformations.

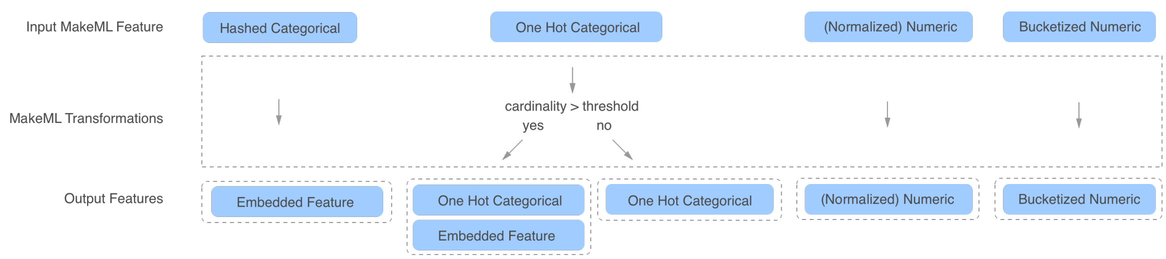

Figure 3-8: MakeML deep feature transformations.

Depending on the hyperparameter configuration, if *include_crossed_columns* is set to true, MakeML will take these base feature and transform them into polynomial cross features. Feature crosses are created for all pairs of features that are of type Bucketized Numeric, Hashed Categorical, or One Hot Categorical.

Finally, MakeML also uses the output features from the string and numeric base feature transformations to build features specific for deep networks. Namely, all hashed columns and categorical columns with cardinality above a given threshold are converted into trainable embedding features. See Figure 3-8.

In the next section, we describe these transformations in greater detail and we provide motivating examples for why these transformations are important in creating high-performing models.

3.7.1 Feature Transformations

Categorical Data

Encoding Machine learning models operate on numeric data. For example, in the Titanic dataset, we are given features such as gender encoded as the strings `Male` and `Female`. We cannot feed these strings directly into a machine learning classifier. We must transform them into a format that machine learning algorithms will understand. In the case of a binary variable, we can encode `Male` as 0 and `Female` as 1. For a categorical variable with more than 2 options, such as the `Embarked` feature with 3 categories, S, C, and Q, if we used the encoding $S \rightarrow 0$, $C \rightarrow 1$, and $Q \rightarrow 2$, we would be assuming an implicit ordering $S < C < Q$. This would be a poor encoding choice. Instead, we encode these categorical features using a one-hot encoding as shown in the mapping below:

$$\begin{aligned} S &\rightarrow [1, 0, 0] \\ C &\rightarrow [0, 1, 0] \\ Q &\rightarrow [0, 0, 1] \end{aligned} \tag{3.1}$$

Hashing If a categorical feature has many categories, the previous encoding scheme is not memory efficient. For example, in the Titanic dataset, there are 661 different last names out of 891 different passengers. In the one-hot encoding scheme mentioned above, we would need a 661-dimensional vector to represent each of these features. For high dimensional categorical features, we instead use a hashed representation in which we can control the dimensionality of the vector representing the feature. If we were to represent last names as a 100-dimensional vector, hashing each last name would give us an index into this 100-dimensional vector. We lose some information because individuals with different last names will be hashed to the same index, however, hashing has been shown to work well in practice [12].

Embedding Another possibility for transforming a categorical column is to use an embedding. An embedding is a representation of a features as a vector of real numbers. When training a linear model, we convert `last_name` using a pre-trained embedding, and when training a deep model, we initialize the embedding with a pre-trained word embedding or with random weights and then train this embedding along with the other weights in the neural network. For more details on word embeddings, see [10] [28]. MakeML uses a commonly used heuristic to select the dimension of the embedding layer: $\sqrt[4]{categrories}$ [12].

Numerical Data

Standardization Another common trick used in feature engineering is to normalize numeric features in order to improve performance and trainability. MakeML standardizes all numeric columns to have mean 0 and standard deviation of 1.

Binning A trick often used with numeric data is to bucketize a feature into discrete groups. Consider the age column in the Titanic dataset. Binning allows our linear model to learn a nonlinear dependency of survival rate on age. Binning also helps with the problem of skew in the distribution of numeric data which can adversely effect model performance. MakeML splits numeric features into bins in addition to using the raw numeric feature. The binning strategy used is very naive, where all numeric features are split into 10 equal width bins. The quartile binning strategy is an alternative binning strategy by which data is binned based on the quartiles of the distribution.

Feature Crosses When using linear models, it is often helpful to include nonlinear features as discussed in the section on Feature Binning. Feature crosses are another strategy used to introduce nonlinearities in linear models. They allow us to capture relationships between several different base features. The example provided in [12], describes the benefit of creating such features for latitude and longitude pairs. Each of latitude or longitude alone given to a linear model is not nearly as useful as a grid of latitude, longitude pairs. In the case of the Titanic dataset, the combination of age and gender is a good example of a feature cross that may improve performance. This type of feature transformation is often referred to as Basis Function Regression. [38].

$$y = a_0 + a_1x_1 + a_2x_2 + ...$$
$$y_{poly} = a_0 + a_1x_1^2 + a_2x_2^2 + a_3x_1x_2 + ...$$

The first equation represents a linear model without crossed features. The second equation represents a linear model with a feature cross between x_1 and x_2 where a separate weight, a_3, is learned.

3.7.2 Deep Model Architecture

There are a number of heuristics available for determining the number of hidden units and hidden layers one should use for a deep model [20]. Choosing these parameters is very important because the network architecture can largely determine whether a model will overfit or underfit. Furthermore, too many hidden units will require long training times.

The following heuristics are provided in [20]:

1. The number of hidden neurons should be between the size of the input layer and the size of the output layer.

2. The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.

3. The number of hidden neurons should be less than twice the size of the input layer.

As pointed out in [31], these heuristics ignore the number of training examples and the amount of noise in the targets. Because of this, the author concludes that if not using early stopping or another form of regularization, one must try many different networks, estimate the generalization performance of each, and choose the best one [31]. Heeding this advice while also trying to encode some of the heuristics provided, we develop a randomized strategy for choosing the architecture of the network. For each deep network tried, we choose the number of layers l_i to be randomly drawn from $\{2, \ldots, N\}$, where N is naively chosen to be 4 during our benchmarking but could be chosen to be any integer greater than 1. That is:

$$l_i \sim \{1, \ldots, N\} \tag{3.2}$$

In order to keep the number of hidden units and thus the number of trainable weights in the network in a reasonable range as compared to the number of training examples, we use the following strategy for selecting the number of hidden units in each hidden layer: the number of hidden neurons should be less than twice the size of the input layer.

Since it's possible to have up to 4 hidden layers, and we want to obey the heuristic that the number of hidden neurons is less than twice the size of the input layer, we choose the number of units in a given hidden layer l_i to be the following:

$$hidden\_units_{l_i} \sim \left\{ \lfloor \frac{1}{3} * input\_size) \rfloor, \ldots, \lfloor \frac{2}{3} * input\_size \rfloor \right\} \tag{3.3}$$

3.8 Results

Given a set of trained models, we are left with the task of determining the best model. There are several metrics to consider when evaluating model performance and the best metric could be task specific. One of the most common metrics for classification tasks is to use the `Accuracy` score. One of the biggest problems with the accuracy score is that it is only suitable for problems where there are an equal number of observations in each class and all errors in prediction are equally weighted. For a more detailed

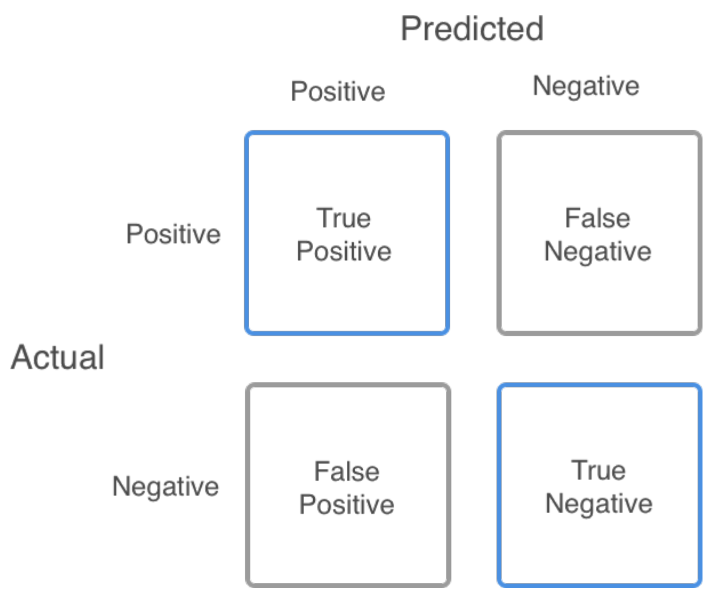

Figure 3-9: Confusion Matrix for True/False Positive/Negative definitions.

description of the Accuracy Paradox and how more accurate models can sometimes have worse predictive power, see [13]. The current implementation of model selection in MakeML uses the `Accuracy` score when selecting the best model. Other metrics considered were `Precision`, `Recall`, and `F Score`. See the confusion matrix given in 3-9 for a visual explanation of `True Positive`, `False Positive`, `True Negative` and `False Negative`.

Accuracy

$$\text{Accuracy} = \frac{\text{Number of Correctly Classified Examples}}{\text{Total Number of Examples}}$$

Precision

$$\text{Precision} = \frac{\text{True Positives}}{(\text{ True Positives} + \text{ False Positives})}$$

Recall

$$\text{Recall} = \frac{\text{True Positives}}{(\text{True Positives} + \text{ False Negatives})}$$

F Score (F_β) The F_1 score is the equally weighted harmonic mean between precision and recall and is given by the following equation [1]:

$$F_1 = \frac{2}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

The general form of the F_β score allows for a different weighting of importance of precision or recall and is given by the following equation:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}$$

Chapter 4

User Study

MakeML was given to a group of three individuals who work with data. All members of the test group had familiarity with SQL. These users were observed while using the software and then were asked to fill out a brief survey regarding their experience. The findings from this group of test subjects using *Version 0* of MakeML motivated the changes introduced for *Version 1*.

4.1 Version 0

All users in the test group reported their familiarity with SQL as a 4 or a 5. Of the three users, only one had written code for machine learning, and only two rated their experience writing code as a three or above. All three users reported that they wanted to make predictions on a dataset at one point but that they found programming inaccessible.

One of the biggest shortcomings of the *Builder Interface* emerged in this first study. A user was shown the *Builder Interface* and told to select features to include. Because *Version 0* of the *Builder Interface* had all features as unselected by default, the user only selected a very small subset of features to include in the model. This resulted in a model with very poor performance. Perhaps for other datasets where many features are not relevant, this would be a good approach, but for this particular dataset, all features were relevant. This motivated *Version 1* to have all features

Figure 4-1: Improved Tooltip Explanation for Model Performance Metrics, [1]

selected by default. Because this could be a poor default choice for other datasets, the right approach needs to be considered more closely.

Surprisingly, one of the users who uses SQL daily in their job decided to use the *Builder Interface* instead of the *Query Interface*. One potential explanation is that they were nervous to write SQL in front of an observer and they would have made a different choice had they been given the tool to try in private. This is something that should be explored more.

Overall, 2 of the 3 users found the process of creating features in the *Builder Interface*, "Very straightforward". The other found it "Confusing, but once explained, understandable". Two users said they would use the *Query Interface* in the future and one said they would consider the *Builder Interface* depending on the complexity of the feature engineering. For all three users, configuring and training a model took less than 5 minutes.

Two users found that the explanations for the performance metrics were confusing. This motivated better explanations and a link was added to the tooltip for the user to learn more.

Once users were taken to the performance view, they did not know what action to take. Two users remarked that the results view did not give them an option to run predictions on new data. The first implementation of MakeML included only the training step of model creation. It did not include a step for exploring the predictions. In *Version 1*, a table of predictions from the prediction dataset was added to the results display below the performance metrics.

One user found the description of hyperparameter settings confusing. They did not know how to interpret them and instead wanted to see which features were most important to the trained model. In *Version 1*, for wide and wide & deep models, MakeML displays the feature weights. For the XGBoost models, MakeML displays the feature importances. For deep models, introspection into how they work and which features are important is an open area of research, but tools like LIME [30] could help. In-depth introspection of why a model predicted something is an important feature for the next version of MakeML.

Below are some quotes from the user study:

> "More explanation in the software itself would be helpful" - *Test Subject*

> "I want to see the results!" - *Test Subject*

> "The ability to actually make a prediction using that model (after data has been trained). Would it be possible to put in parameters to do what if analysis? i.e. if the marketing spend goes up by x, what will the impact to sales be?" - *Test Subject*

Chapter 5

Evalutation

In this section, we compare MakeML's performance on the **Titanic: Machine Learning from Disaster** Kaggle Competition with three different models: (1) a model hand-tuned by a professional data scientist, (2) a model trained with Auto-SKLearn in "vanilla mode" (no ensemble and no warm-start optimization) and (3) an Auto-SKLearn model with default configuration. We then look at how different imputation strategies, hyperparameter configurations, and model evaluation strategies impact model performance using the Kaggle Leaderboard Accuracy Score as an evaluation metric. Next, we evaluate MakeML's performance in automated feature engineering and model architecture selection using Tensorflow's Wide & Deep reference implementation as a benchmark [9]. We conclude by comparing MakeML's overall performance against that of Auto-SKLearn using the Pima Indians Diabetes dataset [33] and the Iris dataset [22]. [1]

5.1 Titanic Dataset

We evaluate MakeML's automated feature engineering and model selection against Auto-SKLearn and a solution provided by a professional data scientist using the score achieved on the Kaggle Leaderboard as a performance metric. The best MakeML

[1] Many of the datasets used for benchmarking are available on the UCI Machine Learning Repository, available at [15].

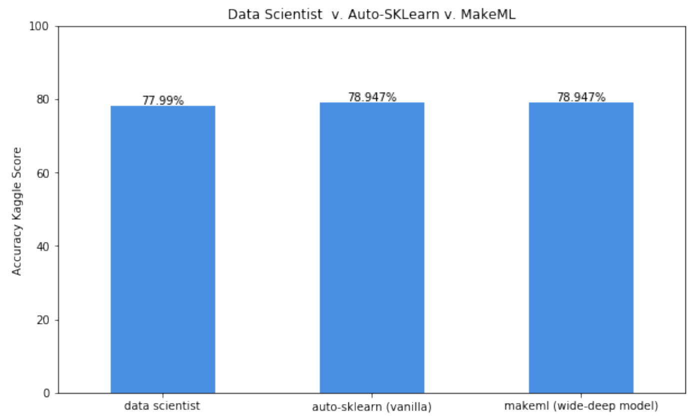

Figure 5-1: MakeML v. Auto-SKLearn v. Data Scientist.

model with hand-selected features using the *Query Interface* achieves an accuracy score of 80.382%, while the best MakeML model without hand-selected features achieves an accuracy score of 78.947%.

Auto-SKLearn In order to evaluate MakeML, we ran two iterations of Auto-SKLearn, one with the default configuration and one using the "vanilla mode" as described in [16], where the ensemble size is set to 1 and the initial configuration via meta learning optimization is disabled [21][2]. For the following evaluations, the same `test` and `train` set are used. The Auto-SKLearn classifier with default configuration achieves an accuracy score of 79.88% on the `test` set, and a score of 78.47% on the Kaggle Leaderboard. The "vanilla mode" Auto-SKLearn classifier achieves a score of 79.30% on the `test` set and a score of 78.95% on the Kaggle Leaderboard.

Data Scientist We gave the Titanic problem to a professional data scientist to see how MakeML would compare. The data scientist implemented a deep model and a random forest model. The features included the base features in addition to the `deck` and `title` features, the same features we were able to create in the *Query Interface*. The random forest was implemented in Python with 116 lines of code and it took the data scientist between about 3 hours to write. The deep model took longer to implement but was eventually discarded due to poor performance. The random forest model achieved an accuracy score of 77.99% on the Kaggle Leaderboard.

quantile	accuracy	auc	precision	recall	f1
0.10	0.733728	0.769152	0.692308	0.390625	0.537634
0.25	0.757396	0.799330	0.740741	0.609375	0.650407
0.50	0.792899	0.845610	0.777778	0.656250	0.706897
0.75	0.810651	0.864807	0.803571	0.687500	0.733333
0.90	0.816568	0.872217	0.841882	0.718750	0.746032
0.99	0.834320	0.880283	0.952381	0.765625	0.771654

Table 5.1: Quantiles of performance metrics across 1205 models trained on Titanic Dataset

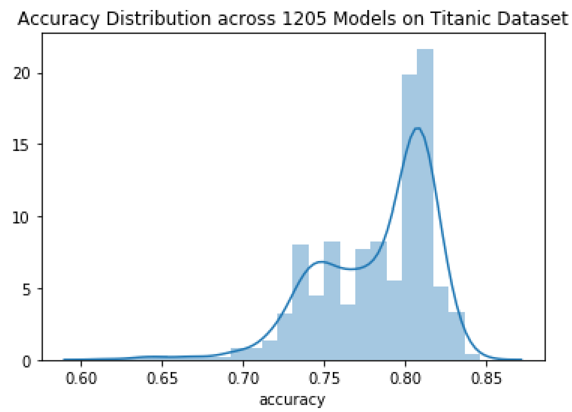

Figure 5-2: Histogram of Accuracy for 1205 models on Titanic Dataset

Accuracy Distribution across Large Hyperparameter Grid Next, we explore the distribution of performance across our entire hyperparameter grid. This gives us a sense of how many training runs result in wasted computation. We use accuracy metrics from a training run using the large hyperparameter grid resulting in 1,205 unique training configurations. A summary of the quantiles and performance cutoffs for each metric is given in Figure 5.1 and a histogram of accuracy across all of these runs is shown in Figure 5-2. Surprisingly, the model with an accuracy score at the 50^{th} percentile of all those tested by MakeML achieves a Kaggle Score of 77.033%.

5.1.1 Effect on Performance of Different Hyperparameter Configurations

We now look at how different hyperparameter configurations impact model performance. The following are results from running MakeML's wide & deep model with

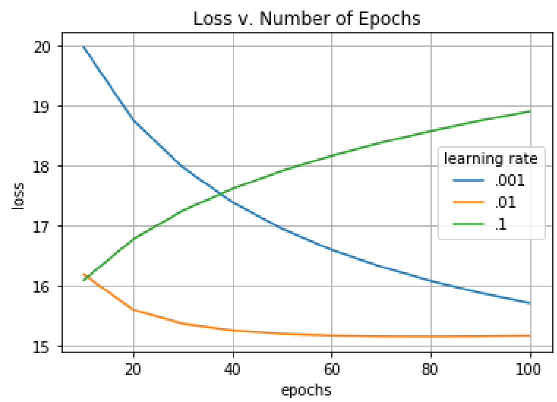

Figure 5-3: Loss v. Number of Epochs for Different Learning Rates

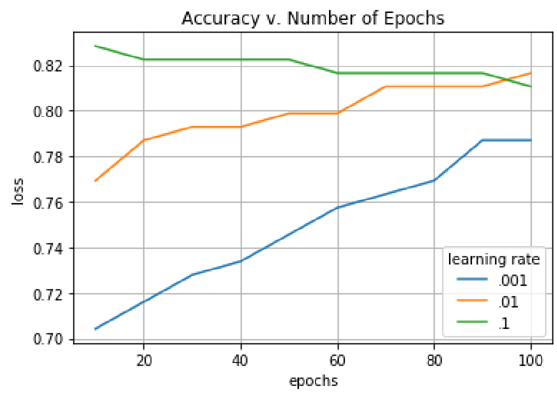

Figure 5-4: Accuracy v. Number of Epochs for Different Learning Rates

L_1 and L_2 regularization set to 0, and k-nearest neighbors (KNN) used as the imputation strategy for both numeric and string columns. The number of hidden units, $[22, 18, 27, 30]$ was chosen using the randomized algorithm described in Figure 3.7.2.

When the learning rate is low, .001, the number of training epochs required to achieve a reasonable score on Kaggle is quite high. The model with 100 epochs gets a Kaggle Score of 68.899%, as compared to the model with 1000 epochs, which gets a score of 73.205%. The model with 700 epochs receives a score of 74.641%. With a learning rate of .001, we need more training epochs in order achieve the highest accuracy score, however, increasing the number of training epochs too much causes the accuracy to decrease again which is an indication of our model overfitting.

With a learning rate of .01, however, we are able to achieve a higher accuracy in a smaller number of epochs. In just 100 epochs, we achieve a Kaggle Score of 78.468%, beating the score achieved from the hand-tuned model by the data scientist. Similarly, a learning rate of .1 and 20 epochs got a Kaggle Score of 78.947% and a learning rate of .1 and only 10 epochs achieves a score of 77.990%, demonstrating that we can achieve a high score for this particular dataset using a modest number of epochs when we have a higher learning rate.

The choice of evaluation metric was incredibly important in the ultimate Kaggle Score for the model. The model with the highest F_1 score got a Kaggle Score of 78.947%, while the model with the lowest loss got a Kaggle Score of 77.511% and the

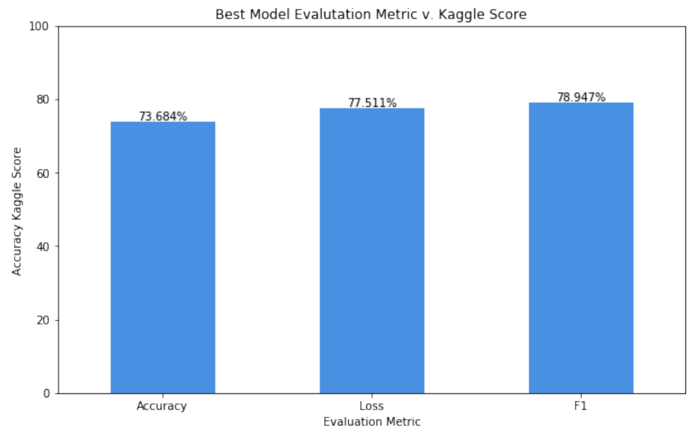

Figure 5-5: Kaggle Accuracy v. Best Model Evaluation Metric

model with the highest accuracy got a Kaggle Score of 73.684% .These findings are summarized in Figure 5-5.

Imputation Strategy Imputation for this particular dataset has only a modest impact on the model's AUC score on the test set. This may not be true for other datasets. The effect of modifying the imputation strategy is show in Figure 5-6.

5.2 Model Architecture and Feature Engineering for Wide & Deep Models

In this section we evaluate MakeML's automated process of feature engineering and deep architecture selection. We use Tensorflow's Wide & Deep Model as a reference implementation [8][9]. The task is to predict whether income exceeds $50k/yr and is a binary classification task. The data consists of 48,842 instances with the following features: `age`, `workclass`, `fnlwgt`, `education`, `education-num`, `marital-status`, `occupation`, `relationship`, `race`, `sex`, `capital-gain`, `capital-loss`, `hours-per-week` and `native-country`. The target labels for prediction are >50K, <=50K [5]. Tensorflow's Wide implementation achieves an accuracy of 83.6% and the Wide & Deep model achieves an accuracy of 85.5%. We ran the reference implementation ourselves to confirm this metric and achieved a score of 85.597%. MakeML's automated feature

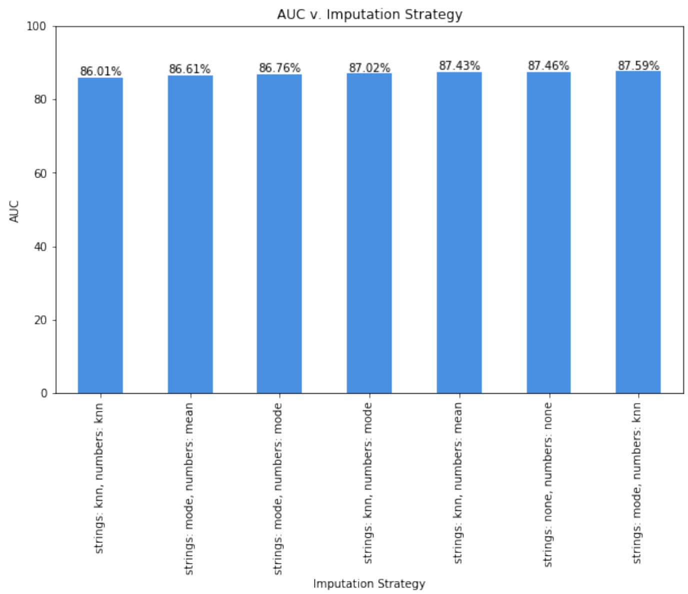

Figure 5-6: AUC v. Imputation Strategy

hidden units	accuracy
$[56, 77, 60]$	86.087
$[77, 59, 57, 58]$	85.921
$[53, 82, 79]$	85.909
$[87, 67]$	85.848
$[67, 55, 50, 70]$	85.768

Table 5.2: Accuracy Across 5 Runs of MakeML Deep Architecture Selection

engineering produces 91 crossed features for a total of 135 features. Using the same learning rate, regularization, and network dimension as the reference implementation, MakeML achieves an accuracy score of 85.96% on the test set. This is good validation that the automated feature engineering in MakeML is competitive with hand-selected features. We then benchmarked MakeML's deep model architecture selection algorithm by comparing the performance of MakeML's deep architecture against the hand-selected hidden units of $[100, 75, 50, 25]$ in the reference implementation [9]. We found that MakeML beat the the reference implementation on all tests, and that MakeML with feature engineering and deep architecture selection produced the model with the highest accuracy, 86.087%.

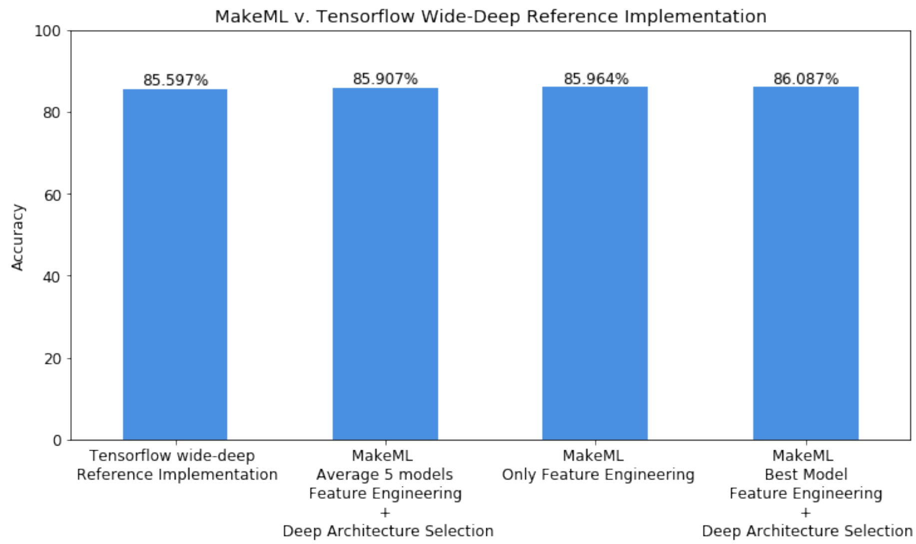

Figure 5-7: Comparison of MakeML Deep Architecture Selection and Automated Feature Engineering to Tensorflow Reference Implementation

5.3 Pima Indians

We also benchmark MakeML using the Pima Indians dataset which consists of 2 classes, 8 attributes, and 768 instances [33]. The task is to predict the onset of diabetes for a group of Pima Indian females. Running 10-fold cross validation on the full dataset, MakeML is able to achieve a mean accuracy across all 10 folds of 76.04%. The best model has the following hyperparameter configuration: `model_type: XGBoost`, `max_depth:4`, `n_estimators:1000`, `learning_rate:0.005`.

We then run Auto-SKLearn using a 80%/20% train/test split. MakeML's XGBoost model achieves an accuracy score of 75.217% on the test set with the following hyperparameters: `max_depth:6`, `n_estimators:300`,and `learning_rate:.005`. The Auto-SKLearn classifier with default configuration achieves an accuracy score of 70.869%. The "vanilla" Auto-SKLearn classifier achieves an accuracy score of 70.435%. If we restrict the time limit allowed for Auto-SKLearn to find the optimal configuration, however, it performs much better, achieving an accuracy score of 75.217% and 75.652% in the default and "vanilla" modes respectively. We also ran a logistic regression with no hyperparameter tuning and $L_2 = 1.0$ and achieved an accuracy score of 72.174%. A chart summarizing these findings is shown in Figure 5-8.

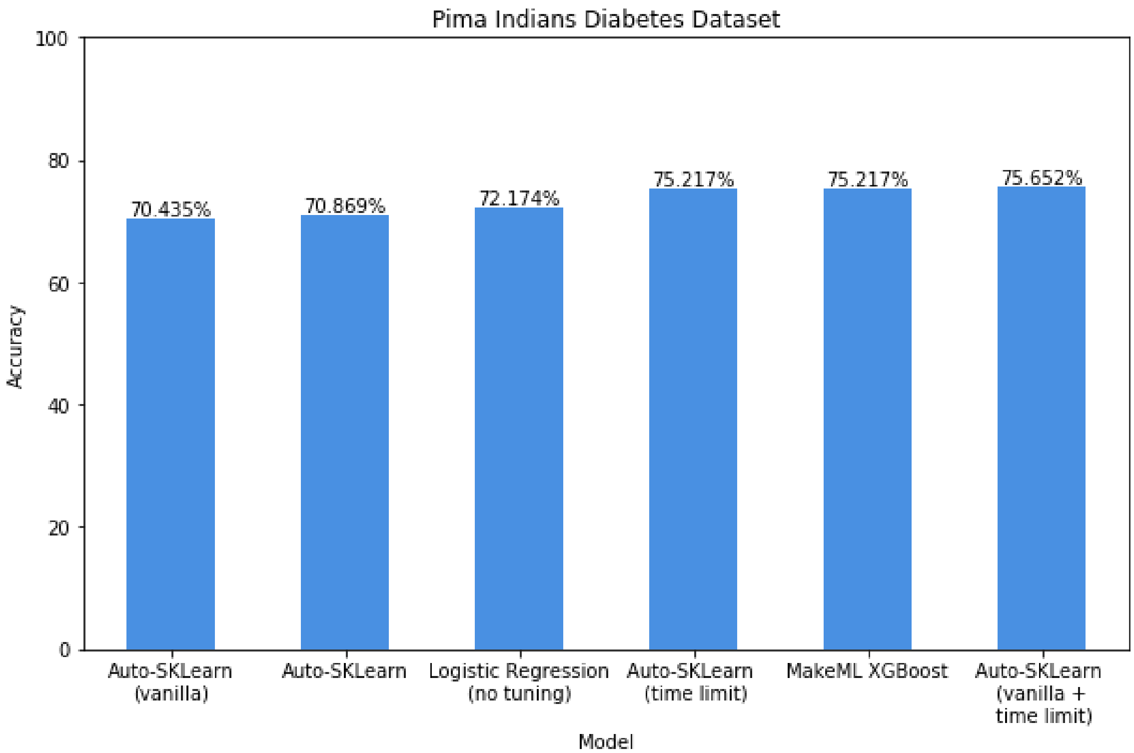

Figure 5-8: Accuracy on the Pima Indians Diabetes Dataset.

5.4 Iris Dataset

The Iris dataset consists of 3 classes of 50 instances each [17]. The features include `sepal length`, `sepal width`, `petal length` and `petal width`. The task is to predict the species of Iris flower, one of `Iris Setosa`, `Iris Versicolour`, or `Iris Virginica`. Running 10-fold cross validation on the full dataset with an XGBoost model, MakeML is able to achieve a mean accuracy across all 10 folds of 95.83%. The best XGBoost model has the following hyperparameter configuration: `max_depth:4`, `n_estimators:10`, `learning_rate:0.001`.

We then compared MakeML with Auto-SKLearn. We split data using an 80%/20% train/test split. The Auto-SKLearn classifier with default configuration achieved a score of 89.80% on the test set, but with "vanilla" Auto-SKlearn, the accuracy is 95.918% on the test set. The XGBoost model from MakeML achieves a score of 94.059% and the MakeML deep model with $L_1 = 0$, $L_2 = 0$, and hidden units $[14, 13, 11]$, gets an accuracy score of 93.88% on the test set. For comparison, if we

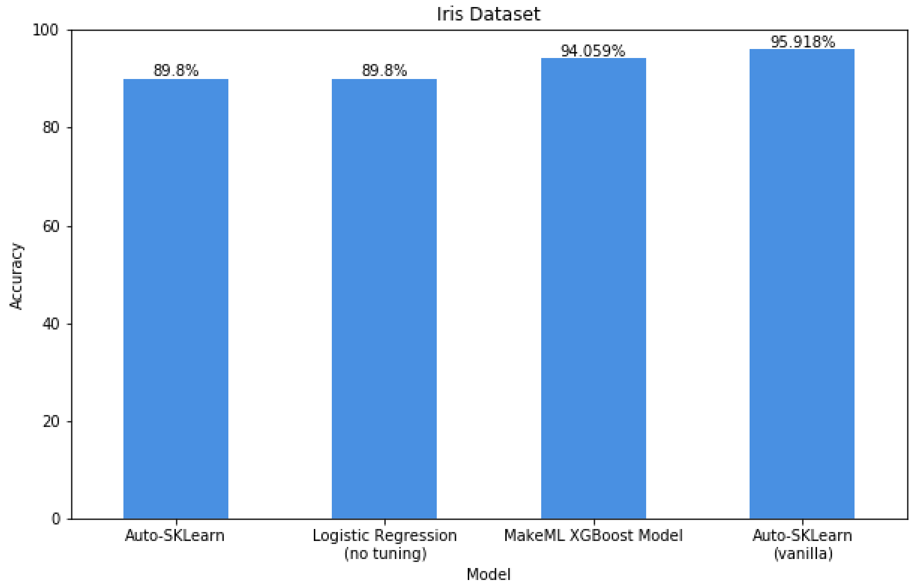

Figure 5-9: Iris Dataset Benchmark

train a logistic regression with no hyperparameter tuning and $L_2 = 1.0$, we get an accuracy score of 89.80% on the test set.

Chapter 6

Related Work

6.1 Auto-Weka

Auto-Weka is a GUI application that allows users to define and train machine learning models [36] [18]. Auto-Weka provides fine-grained control of feature preprocessing, model selection, and hyperparameter tuning. Unlike MakeML, Auto-Weka is only usable by data scientists because it does not abstract away these technical details. Furthermore, it is a single-node desktop application. MakeML, on the other hand, runs machine learning jobs in the cloud on a highly scalable cluster, which allows it to train many large models in parallel.

6.2 Auto-SKLearn

Auto-SKLearn is a high level machine learning library that uses Bayesian optimization to automate the exploration of a number of hyperparamater and feature engineering combinations [16]. Because Auto-SKLearn uses a Bayesian optimization approach to hyperparameter tuning, it is more computationally efficient that MakeML's exhaustive grid search. While Auto-SKLearn automates a portion of the machine learning process, it is inaccessible to non-engineers without programming skills. Further, it does not automatically perform label encoding. For example, Auto-SKLearn cannot take as input the raw titanic dataset with `sex` encoded as `Male`|`Female`. It needs these values to be encoded as numbers. This requires preprocessing code to be written. In contrast, MakeML allows users to train models without a single line of code.

6.3 Data Robot

Data Robot is a commercial product that allows a user to upload a CSV file of their data and automatically run many different models. Like MakeML, DataRobot provides missing value imputation, feature scaling, and one-hot encoding. Unlike MakeML, in which the user is presented with a single best model, DataRobot allows for in-depth comparison of all trained models across a variety of performance metrics, enabling the end user to select the best performing model based on their own performance criteria. Furthermore, DataRobot provides an in-depth interface for introspecting which features are most important to the model. The main distinguishing feature of MakeML is that it allows users to join and explore datasets by connecting directly to the data source, eliminating the need for a two-stage process where data is prepared in a separate tool. Furthermore, MakeML is better suited to work with very large datasets because the data transfer and loading from the relational database is automated, and training is parallelized over a large compute cluster.

6.4 BigML

BigML is a commercial plugin for spreadsheet applications such as Google Sheets that automatically runs machine learning algorithms on data in a spreadsheet. Unlike BigML, MakeML accesses SQL data sources directly, so that large datasets can be trained on without requiring CSV export and data can be joined and aggregated across foreign key relations.

6.5 MLBase

MLBase is a high-level declarative language for specifying machine learning tasks [37]. It automatically trains and optimizes models over a variety of hyperparameter configurations. The result is a learned model that the user can use to make predictions. Unlike MakeML, MLBase provides a more sophisticated strategy for selecting optimal hyperparameter configurations and has the ability to do both supervised and unsupervised learning. While MLBase abstracts the tuning and scalability of machine learning allowing researchers without backgrounds in distributed systems to create machine learning models, MLBase's declarative syntax for creating machine learning jobs still requires the user to write code.

6.6 KeystoneML

KeystoneML is a high level software framework for declaratively writing machine learning pipelines [34]. KeystoneML requires defining preprocessing steps and model configuration precisely. Similarly to MakeML, it automates the horizontal scaling of model training across many nodes. KeystoneML achieves this by using the Apache Spark framework as a backend, while MakeML's implementation is built on top of Pachyderm and Kubernetes. Most importantly, KeystoneML does not abstract away the selection and training of the model, and thus still requires a machine learning engineer.

Chapter 7

Conclusion

In this thesis we have presented the problem of expanding access to machine learning to those without data science skills and described our solution, MakeML. While our initial tests showed promising results, there is a great deal of future work that could be undertaken to improve the performance of the models MakeML can produce and improve the user experience.

Support for Image Data We believe MakeML could cover a larger range of use cases if it provided support for image data. It is very common, for example, for SQL databases to have columns with URLs referencing images. MakeML could support loading images from such sources and training convolutional neural networks to aid in classification or regression.

Explainability In-depth introspection of features is very important, allowing analysts to understand why their model is behaving in a particular way. This is something that came up several times in our user study. In its current form, MakeML has only rudimentary support for model explainability where we display a chart comparing weights for various features of linear models and comparing feature importances for xgboost models, but there is no support for introspection in deep models. For deeper explainability, we could integrate with a system like LIME [30]. This would be even more useful if MakeML supported image data.

Builder Interface Feature Transformations A non-programmer using the *Builder Interface* may be able to provide some hints to the learning algorithm about the structure of data to improve model performance. In the future, we could design a GUI that allows such a user to define simple transformations like this without having to use SQL. Some possible transformations include:

1. The ability to create a feature that is a function of several base features, e.g. $SUM(feature_1, feature_2)$.
2. The ability to perform functions on date fields, like converting a timestamp to the number of days before a supplied `target date`.
3. The ability to do simple text transformations, allowing the analyst to extract substrings of a text field.

Unsupervised Learning, Clustering, Anomaly Detection During user interviews several analysts reported that they would like support for unsupervised learning problems as well as the supervised problems MakeML currently supports. We could add support for automatically clustering unlabeled data or detecting anomalies.

Appendix A

Appendix

A.1 AutoML

Automated Machine Learning stems from the idea that many common tasks of a machine learning engineer involve processes that can be automated [27]. For instance, when trying to find an optimal solution for a given prediction task, a machine learning engineer will often try many different algorithms and many different hyperparameter configurations for each of these algorithms. Furthermore, different algorithms have different expectations for the encoding of input features. All of this hyperparameter search and configuration can be encapsulated into an automated system, freeing the machine learning engineer from the repetitive processes that are a time consuming component of their daily work. The AutoML problem is defined in [16] as follows:

> **AutoML** For $i = 1, \ldots, n+m$, let $x_i \in R_d$ denote a feature vector and $y_i \in Y$ the corresponding target value. Given a training dataset $D_{train} = (x_1, y_1), \ldots, (x_n, y_n)$ and the feature vectors $x_{n+1}, ..., x_{n+m}$ of a test dataset $D_{test} = (x_{n+1}, y_{n+1}), \ldots, (x_{n+m}, y_{n+m})$ drawn from the same underlying data distribution, as well as a resource budget b and a loss metric $\mathcal{L}(\cdot, \cdot)$, the AutoML problem is to (automatically) produce test set predictions $\hat{y}_{n+1}, \ldots, \hat{y}_{n+m}$. The loss of a solution $y_{n+1}, \ldots \hat{y}_{n+m}$ to the AutoML problem is given by $\frac{1}{m}\sum_{j=1}^{m}\mathcal{L}(\hat{y}_{n+j}, y_{n+j})$

A.2 Hyperparameter Optimization

A.2.1 Hyperbandit

The Auto-SKLearn Bayesian optimization approach is sequqential and difficult to parallelize. Hyperband: A Bandit-Based Approach to Hyperparameter Optimization, introduced by Li et.al., is able to achieve five to thirty times better speedup over state-of-the-art Bayesian optimization algorithms by formulating the problem of finding the optimal hyperparameter configuration as a multi-armed bandit problem [26].

A.2.2 Population Based Training

Yet another approach to hyperparameter optimization and model search called Population Based Training was introduced by Jaderberg et. al.[24]. The population based training algorithm addresses one of the biggest shortcomings of grid search, in which only a small number of models are trained with good hyperparameters [23]. It also addresses the problem of Bayesian optimization and hand-tuning in which each successive training run must wait for the previous training run to complete in order to determine the next hyperparameter configuration to test. The Population Based Training algorithm starts by creating many workers, each training a model with a random hyperparameter configuration. During training, each of these models can either copy the hyperparameter configuration from the currently best-performing model in the population, randomly select new hyperparameters, or continue training with its current configuration [23].

A.3 Pachyderm and Kubernetes

Kubernetes is an open-source container orchestration system [4]. Given a description written in the Kubernetes DSL of a number of computation tasks packaged in Docker containers, a small number of Kubernetes master nodes will schedule the tasks to run on a large number of worker nodes. The Kubernetes software ensures that the correct number of containers are run and that they are scheduled on nodes to maximize the use of computational resources. At the time of this writing, Kubernetes has been tested to reliably schedule containers on clusters of up to 5,000 nodes. The battle-tested scalability of Kubernetes is what enables MakeML to perform well on very large datasets and with many concurrent jobs.

Kubernetes excels at ensuring the execution of containerized jobs but it does not provide the ability to define dependencies between jobs to form a pipeline. To provide this functionality, we employ Pachyderm, an open-source data science pipeline and data management system built on top of Kubernetes [6]. Pachyderm consists of two components: the Pachyderm File System (PFS) and the Pachyderm Pipeline System (PPS).

PFS is a distributed file system with semantics similar to the git version control system. Clients of PFS can manage a group of repositories, each containing a series of commits identified by content hashes. Individual commits can be queried to list and read files, and new commits can be added to repositories to create of modify files. PFS is built for massive data-set sizes because data storage is provided by an Amazon S3 compatible storage backend such as Amazon S3, Google Cloud Storage, or one of many open source alternatives.

PPS manages multi-stage data science pipelines. Each pipeline stage consists of a job to be run packaged in a Docker container and a description of the pipeline stage defined in the Pachyderm DSL. The pipeline specification indicates the PFS repository to input to the job, the name of the pipeline, the code to run and the parallelism spec which specifies the number of workers to start for a given job [3]. For a full description of the configuration options of a pipeline, see [7]. The configuration of the **Hyperparams** pipeline stage is shown in Figure A-1.

When the job runs, the input files will be mounted in the container at the path `/pfs/<repository_name>` and the output repository will be mounted at `/pfs/out`. The implementation of the job may simply read files from `/pfs/<repository_name>` and write files to `/pfs/out` using standard filesystem operations of the operating system. PPS ensures that these file operations read and write the appropriate data from the PFS storage backend. PPS watches for new commits to the input repository

```json
{
  "pipeline": {
    "name": "hyperparams"
  },
  "transform": {
    "image": "makeml/hyperparams",
    "cmd": [ "/bin/sh" ],
    "stdin": [
      "python3 -u main.py --input /pfs/merge --output /pfs/out"
    ]
  },
  "enable_stats": true,
  "input": {
    "atom": {
      "name": "merge",
      "repo": "merge",
      "glob": "/*",
      "lazy": true
    }
  }
}
```

Figure A-1: Pachyderm pipeline configuration for **Hyperparams** pipeline stage

and schedules containers to run on Kubernetes whenever a new commit is added. If there are many files in the commit, Pachyderm will schedule a large number of containers to run, based on the parallelism spec defined in the pipeline configuration file [7]. When a job adds a commit to its output repository, all pipeline stages bound to that repository will be run. In this way, PPS pipeline stages form a directed acyclic graph of computation and data flow with high parallelism. All MakeML machine learning processes are implemented as a series of PPS pipeline stages.

Bibliography

[1] 3.3. model evaluation: quantifying the quality of predictions. http://scikit-learn.org/stable/modules/model_evaluation.html#precision-recall-f-measure-metrics.

[2] Auto-sklearn manual. Available: https://automl.github.io/auto-sklearn/stable/manual.html.

[3] Creating analysis pipelines. Available: http://pachyderm.readthedocs.io/en/latest/fundamentals/creating_analysis_pipelines.html.

[4] Kubernetes version 1.8.8. Available: https://github.com/kubernetes/kubernetes.

[5] Machine learning respository: Census income data set. Available: https://archive.ics.uci.edu/ml/datasets/Census+Income.

[6] Pachyderm. Available: https://github.com/pachyderm/pachyderm.

[7] Pachyderm pipeline specification. Available: http://pachyderm.readthedocs.io/en/latest/reference/pipeline_spec.html. Revision: 6b067d8d, https://github.com/pachyderm/pachyderm/blob/master/doc/reference/pipeline_spec.md.

[8] Tensorflow linear model tutorial. Available: https://www.tensorflow.org/tutorials/wide.

[9] Tensorflow wide and deep learning tutorial. Available: https://www.tensorflow.org/tutorials/wide_and_deep.

[10] Vector representations of words. Available: https://www.tensorflow.org/tutorials/word2vec.

[11] 10 key marketing trends for 2017. Available: https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WRL12345USEN, 2016.

[12] Feature columns. Available: https://www.tensorflow.org/get_started/feature_columns, 2018 (accessed Fri, 11 May 2018). Tensorflow Organization.

[13] Jason Brownlee. Classification accuracy is not enough: More performance measures you can use. Available: https://machinelearningmastery.com/classification-accuracy-is-not-enough-more-performance-measures-you-can-use/, March 2014.

[14] Jason Brownlee. How to handle missing data with python. Available: https://machinelearningmastery.com/handle-missing-data-python/, March 20, 2017 (accessed Fri, 11 May 2018).

[15] Dua Dheeru and Efi Karra Taniskidou. UCI machine learning repository, 2017.

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

[17] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(7):179–188, 1936.

[18] Eibe Frank. Machine learning with weka. Available: https://sourceforge.net/projects/weka/.

[19] T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer, 2009.

[20] Jeff Heaton. The number of hidden layers. Available: http://www.heatonresearch.com/2017/06/01/hidden-layers.html.

[21] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, page 507âĂŞ523, 2011.

[22] Iris data set. Available: https://archive.ics.uci.edu/ml/datasets/iris.

[23] Max Jaderberg. Population based training of neural networks. Available: https://deepmind.com/blog/population-based-training-neural-networks/, November 2017.

[24] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *CoRR*, abs/1711.09846, 2017.

[25] Titanic: Machine learning from disaster. Available: https://www.kaggle.com/c/titanic.

[26] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560, 2016.

[27] Matthew Mayo. The current state of automated machine learning. Available: `https://www.kdnuggets.com/2017/01/current-state-automated-machine-learning.html`, Jan 2017.

[28] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[29] Andrew Ng. Part vii: Regularization and model selection. Available: `http://cs229.stanford.edu/notes/cs229-notes5.pdf`. CS229 Lecture Notes.

[30] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should I trust you?: Explaining the predictions of any classifier. *CoRR*, abs/1602.04938, 2016.

[31] Warren S. Sarle. How many hidden units should i use? Available: `http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-10.html`. (accessed Fri, 25 May 2018).

[32] Cross-validation: evaluating estimator performance. Available: `http://scikit-learn.org/stable/modules/cross_validation.html#cross-validation`.

[33] Everhart J. E. Dickson W. C. Knowler W. C. Smith, J. W. and R. S. (1988). Johannes. Using the adap learning algorithm to forecast the onset of diabetes mellitus. In *Proceedings of the Annual Symposium on Computer Application in Medical Care*, pages 261–265. IEEE Computer Society Press.

[34] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. *CoRR*, abs/1610.09451, 2016.

[35] LinkedIn Economic Graph Team. Linkedin's 2017 u.s. emerging jobs report. Available: `https://economicgraph.linkedin.com/research/LinkedIns-2017-US-Emerging-Jobs-Report`, December 2017.

[36] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proc. of KDD-2013*, pages 847–855, 2013.

[37] John Duchi Rean Griffith Michael Franklin Tim Kraska, Ameet Talwalkar and Michael Jordan. Mlbase: A distributed machine-learning system. In *In Conference on Innovative Data Systems Research*, 2013.

[38] Jake VanderPlas. Python data science handbook: Essential tools for working with data. Available: https://jakevdp.github.io/PythonDataScienceHandbook/05.06-linear-regression.html, November 2016.