# PALM: Machine Learning Explanations For Iterative Debugging

Sanjay Krishnan
UC Berkeley
sanjaykrishnan@berkeley.edu

Eugene Wu
Columbia University
ewu@cs.columbia.edu

## ABSTRACT

When a Deep Neural Network makes a misprediction, it can be challenging for a developer to understand why. While there are many models for interpretability in terms of predictive features, it may be more natural to isolate a small set of training examples that have the greatest influence on the prediction. However, it is often the case that every training example contributes to a prediction in some way but with varying degrees of responsibility. We present Partition Aware Local Model (PALM), which is a tool that learns and summarizes this responsibility structure to aide machine learning debugging. PALM approximates a complex model (e.g., a deep neural network) using a two-part surrogate model: a meta-model that partitions the training data, and a set of sub-models that approximate the patterns within each partition. These sub-models can be arbitrarily complex to capture intricate local patterns. However, the meta-model is constrained to be a decision tree. This way the user can examine the structure of the meta-model, determine whether the rules match intuition, and link problematic test examples to responsible training data efficiently. Queries to PALM are nearly 30x faster than nearest neighbor queries for identifying relevant data, which is a key property for interactive applications.

**ACM Reference format:**
Sanjay Krishnan and Eugene Wu. 2017. PALM: Machine Learning Explanations For Iterative Debugging. In *Proceedings of HILDA'17, Chicago, IL, USA, May 14, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3077257.3077271

## 1 INTRODUCTION

Machine learning (ML) is an integral part of many software systems such as fraud detection, content recognition, and automation. This increased reliance on ML leads to the crucial need to develop tools to debug and explain these models in order to understand failure cases and further improve their accuracy. In the past, ML models were often highly structured (e.g., decision trees are constrained to axis-aligned splits), and this structure could be used for debugging (e.g., whether the splits match intuition of how variables relate to the labels). However, a recent trend is to use deep neural networks, whose structure is more complex and difficult to directly interpret. This results in black-box models that render existing ML evaluation, debugging, and interpretation strategies ineffective. While the database community has made substantial contributions in other areas of machine learning (e.g., training scalability and model specification [1, 2, 5, 8, 10]), the opportunity is ripe for developing scalable tools to debug modern machine learning models.

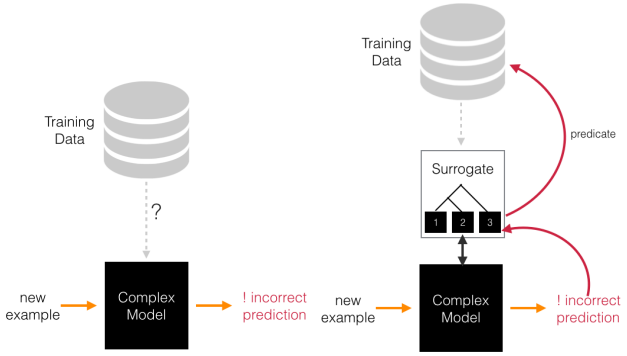


**Figure 1: (Left) ML developers need to understand the relationship between training data and a model's predictions, i.e., how are similar points predicted in the training data. But effectively determining the size and range of this neighborhood is challenging in expressive models which can have very complex geometry. (Right) We propose an approximation technique that can represent a black-box model in terms of sub-models applied to feature-space partitions. The developer can quickly determine which data are related based on these partitions.**

One approach is to explain a complex model using a simplified approximation (a *surrogate model*) [13, 14, 16]. For example, we could identify a subset of the training data in the neighborhood of a mis-predicted test record, and fit a sparse linear model. [13, 14, 16]. Although these approximations can identify salient features, the developer might also want to select a subset of training examples most responsible for the prediction. This could help the developer answer questions such as: (1) what training data most influenced this prediction? (2) how different was this training data from the test example? and (3) where the differences were due to genuine variation or systematic errors in the training data? Answering these questions can help the developer focus on cleaning the training dataset, gather more training data, or augmenting the model.

For these reasons, we propose a new model explanation technique that helps explain a model in terms of partitions of the training dataset. These partitions reflect how training examples influence a prediction from the model. By influence, we mean that if the labels or features were perturbed for this subset, it would be more likely to change the prediction. This notion is similar to responsibility in the data lineage and explanation literature [4, 17], in the context of


*HILDA'17, Chicago, IL, USA*

DOI: http://dx.doi.org/10.1145/3077257.3077271

black-box machine learning models as opposed to database queries with well defined semantics. The problem in machine learning is more challenging because every training data point has some impact on every prediction. High-dimensional, non-linear models, such as Neural Networks, have complex decision boundaries, which makes a simple nearest neighbor search unreliable. Thus, we have to identify a subset of data that aligns with the decision structure of model.

We present Partition Aware Local Model (PALM), which approximates a complex model (e.g., a deep neural network) using a two-part *surrogate model*: a meta-model that partitions the training data, and a set of sub-models that approximate the patterns within each partition. These sub-models can be arbitrarily complex to capture intricate local patterns. However, the meta-model is constrained to be a decision tree. This formulation ensures that sub-models are accurate, while the partitions are interpretable as rules. In contrast to existing model interpretation techniques that return locally relevant features, our approach returns the most informative neighborhood (partition).

Applying PALM to a debugging task is very simple—if a test point is mispredicted, then we can identify the partition it belongs to and investigates that subset of the training data and its associated sub-model. As our experiments show, the developer can tune the total number of sub-models and the depth of the decision tree to control the trade-off between the specificity of the partitions and how similar the surrogate model is to the complex model. We anticipate that PALM is a starting point for the developer to further explore the training data and more quickly identify the problematic training data using other visualization or data summarization tools.

The rest of this short paper describes the PALM problem formulation, an algorithm sketch for training surrogate models, as well as an illustration of how PALM can be integrated into a visual model building interface. Our experiments show that we can generate Fast And Clear Explanations using PALM. We show that the explanations found by PALM select subsets of data that better align with the structure of the original model than the nearest neighbor or clustering baseline. Furthermore, we show that PALM is nearly 30x faster than a nearest neighbor query, which is the next most accurate baseline.

## 2 FRAMEWORK AND API

Given a dataset with a mix of "interpretable" (structured features) and "uninterpretable" (unstructured / high-dimensional features), PALM uses an EM-like algorithm to learn a set of rules over the interpretable features that partition the predictive behavior of the model. These partitions can be used to isolate subsets of training data that affect the prediction of a new test example.

### 2.1 Notation

Let the training dataset $D$ contain tuples of features $x_i$ and labels $y_i$ that are categorical or real-valued. `train(D)` is a training algorithm that returns a model `model(·)` that takes as input a test point $x_{new}$ and predicts a label $\hat{y}_{new}$:

$$\hat{y}_{new} = \mathsf{model}(x_{new})$$

Consider the following running example of car insurance fraud detection. The training dataset $D$ has the following schema:

```
D(make, amount, at_fault, descr, fraud?)
```

where `make` is a categorical attribute describing the make of the car, `amount` is a double-valued amount that the person is claiming, `at_fault` is a boolean variable describing whether the claimant is at fault, `descr` is a string-valued attribute describing the nature of the claim, and `fraud?` is the yes/no label if the claim is fraudulent.

Suppose model(·) issues an incorrect prediction and erroneously flags a fraudulent claim as not fraudulent. The data scientist is now tasked with debugging the model to understand why this error happened. If she was using a simple model, like a decision tree, she might be able to look at the structure of the model and determine whether it matches intuition (e.g., fraudulent claimants tend not to claim they were at fault). However, sometimes the prediction problem necessarily needs a more complex learning model. For example, there is a textual field descr, which might be a very valuable feature for predicting fraud. Processing such data typically requires translating the data into a higher-dimensional feature space by using NLP methods such as word-embeddings, stop word removal, bi-gram featurization, etc. By design, this feature space may not be interpretable by anyone other than a language expert, and using a simpler model may not achieve the desired accuracy. Furthermore, highly expressive deep models such as deep neural networks, which in principle can learn any deterministic function, are susceptible to adversarial examples (i.e., imperceptible perturbations to the features that cause a change in prediction)[15]. This means that relying on neighboring points can be unreliable or even misleading because they may inadvertently be adversarial.

### 2.2 Debugging with PALM

The approach that we propose identifies records that the *classifier* considers similar, rather than naive similarity in the feature space. As an illustrative example, imagine if we hard-coded the following logic:

```
def smodel(x):
    if amount > 10000:
    #one model for larger claims
        return submodel_1(x)
    elif a_fault:
    #one model for small at fault claims
        return submodel_2(x)
    else:
    #a default model
        return submodel_3(x)
```

In this function, interpretable rules assign new data to submodels (which encapsulate the complexity). Although the rules are simple, the full function `smodel` can still model complex patterns due to the complex sub-models. If we observe an anomaly, the developer can precisely blame one of the submodels; thereby, providing a coarse predicate to select tuples that are assigned to that model. Furthermore, the rules illustrate the implicit partitions and decision structure of the function.

Given this intuition, PALM explores whether we can synthesize such code structure automatically to mimic black-box (but differentiable) models. An interpretable meta-model over the structured features (e.g., make, amount, at_fault), assigns a test example to a more complex sub-model over all features. The interpretable meta-model is represented as a decision tree (if-else statements), and the

sub-model is allowed to be of the same class as the original model. In the inference procedure, we learn the most likely decision tree over the interpretable features a set of sub-models that explain a models predictions. There is an inherent tradeoff, a meta-model that contains a single partition `true` can create a sub-model that is nearly identical to the original complex model. However the partition is not useful to the developer. In contrast, increasing the complexity of the meta-model makes each partition smaller, but the resulting model potentially diverges from the original model. At the limit, the meta-model may simply build a decision tree over the training data, and each partition contains nearly identical points. We present initial results exploring these tradeoffs in the experiments.

The inference algorithm can run offline during the training phase and is no-more than a constant factor more expensive than standard model training We presented a generalization in prior work [6, 11]. At a high-level, the algorithm takes the dataset $D$ and the model model as input, and returns smodel, which consists of a decision-tree meta model that selects from a collection of $k$ submodels. Given a new record, the user can evaluate both:

$$\hat{y}_{new} = \mathsf{model}(x_{new})$$

$$\hat{y}_{new} = \mathsf{smodel}(x_{new}) \approx \mathsf{model}(x_{new})$$

and use the structure of smodel to debug with knowledge that it approximates model.

## 2.3 API and System

Our current implementation is in Python and focused on TensorFlow models. The user supplies:

- *Featurized Dataset.* The user provides a dataset of feature and label tuples.
- *Explainable Features.* The user lists a subset of features that are understandable.
- *Tensorflow Model Description.* The user provides a symbolic description of the model in Tensorflow.
- *Number of Sub-models.* The user provides the number of submodels to include in the surrogate model (denoted as $k$).

The output of the system is:

- *Original Model.* The original model trained to completion
- *K Sub-Models.* The system returns K submodels trained on different partitions of the feature-space.
- *Decision Tree Meta Model.* The system returns a meta model that switches between the K submodels based on the input record.

# 3 ALGORITHM DESCRIPTION

In prior work, we designed an algorithm in a completely different context—to decompose complex control policies to reduce planning horizons [6, 11]. Surprisingly, when we started discussing our ideas with ML developers, we realized a very similar approach could apply for problems in debugging.

For intuition on how the algorithm works, consider a standard KMeans clustering of dataset. First, $k$ random cluster centers are placed over the feature-space. Then, data are assigned to the nearest cluster. Finally, the clusters are updated based on the assignment, and the algorithm repeats by re-assigning data based on the most recent update.

Similarly, we can do the same thing for model training. $k$ random sub-models are initialized. During the "assignment" step, a data point is assigned to the sub-model that best predicts it. Then, during the update step, the models are updated based on the new assignments with gradient descent. This is a variant of the popular Expectation-Maximization algorithm, that instead of a Maximization step computes a gradient instead.

This algorithm to run efficiently at scale and directly integrate with TensorFlow's Python API. So, at training time (offline) the user can not only train the original model but also the surrogate. We also integrated the algorithm into a local web interface that visualized the results.

## 3.1 Technical Details

**Fitting Step:** To construct smodel$(\cdot)$ from model$(\cdot)$, we first start off by fitting the parameters to a simplified probabilistic model. The first step is to run model$(\cdot)$ over the entire training dataset and get feature-prediction $(x, \hat{y})$ tuples. We can define $f(\hat{y} \mid x)$ which is the probability of the label $\hat{y}$ given the feature $x$ to represent how model$(\cdot)$ generates predictions. Arbitrary probability distributions are hard to reason about so we consider parametrized distributions $f(\hat{y} \mid x, \theta)$. We want to find $k$ such distributions that best explain all the observations:

$$\{f(\hat{y} \mid x, \theta_1), ..., f(\hat{y} \mid x, \theta_k)\}$$

Our results in prior work [6] show how this can be optimized with a two-step algorithm:

- Initialize $\theta_1, ..., \theta_k$ randomly.
- Repeat until convergence
- For each data point $i \in \{1, ..., N\}$:
  - For each component distribution $j \in \{1, ..., k\}$:
  - $w(i, j) = f(\hat{y}_i \mid x_i, \theta_j)$
- Gradient Ascent for each $\theta$:

$$\theta_j \leftarrow \theta_j + \lambda \sum_i^N w(i, j) \nabla \log f(\hat{y}_i \mid x_i, \theta_j)$$

The intuition behind this algorithm is that after random initialization the initial models will have higher accuracy on different data points (by chance). $w(i, j)$ is a soft-assignment–assigning data points to the model that best explains its prediction. Then, $w(i, j)$ becomes a weight to update the models with Gradient Ascent (or descent over the negative log likelihood). This process repeats until convergence. This is very similar to the K-Means or EM algorithm, but instead of updating the cluster centers with a formula, we take a gradient step.

**Distillation Step:** The result of the first step is a set of model parameters $k$ $\theta_1, ..., \theta_k$, and a weighting function $w(i, j)$. The next step is to distill $w(i, j)$ into a set of explainable rules that select the one of the $k$ models. We first generate a set of hard assignments for each data point:

$$h(i) = \arg\max_j w(i, j)$$

Each $h$ is an indicator $1, ..., k$ of the most likely assignment of each data point. We can now train a more explainable model to select $h$ as

a function of the data. This intuition is that this is a simpler model that just selects one of the component models.

The user provides us with a list of features that are considered "explainable", and let $x_e$ denote the projection of an example onto this subset of features. Then, we can train a decision tree over the tuples $(x_e, h)$, which have the desirable property of resembling programmatic statements. This decision tree is a multi-class classifier that predicts the assignment to a submodel as a function of the intrepretable features. We call this model the meta-model, as it selects between the component models. Finally, putting everything together, we return something that looks similar to the hard-coded example in the previous section. The surrogate model $\mathsf{smodel}(\cdot)$ encapsulates submodels that apply in different parts of the feature-space. Suppose, we observe an anomaly, we can now blame a specific model and efficiently get a predicate that selects all of the data the contributed to the model.

## 4 HIGHLIGHTED EXPERIMENTS

This section presents key experimental results that highlight use cases where PALM models can help identify training data that cause mispredictions, when PALM diverges from the user's more complex model, and a comparison with alternative explanation approaches.

### 4.1 Setup

We consider the following scenario. Suppose, the model receives a previously unseen test point that is mispredicted, can we identify the subset of training data that "influences" this prediction error. PALM returns a subset of training data using the hierarchical approximation, but we could, in principle, apply the following approaches to select relevant training data:

**Nearest Neighbors:** Suppose, we measure influence purely in terms of similarity to the new data point. We could select the k-nearest neighbors in the training dataset and return them to the user. We set $k$ to be the same as number of data points returned by PALM for the new data point.

**Clustering:** Another approach would be to cluster the training examples, and then return the cluster closest to the new data point. We use K-Means and set the number of clusters to 10.

**Random:** Finally, as a baseline, we could randomly select training examples. We set the number of selected examples to be the same as number of data points returned by PALM for the new data point.

We measure influence in the following way. If a sample of labels of the training data outside of the selected subset are randomly perturbed, and the model is retrained, how often does the prediction change. In other words, this is a measure how "true" the selected neighborhood is w.r.t how the model implicitly partitions the feature space.

### 4.2 Datasets

We split each dataset into a training dataset 80% and test dataset 20%. **Movie:** We have a dataset of movie descriptions IMDB [1] and Yahoo [2]. Each movie has a title, a short 1-2 paragraph plot description, year, rating, language, and a list of categories, and the goal is to train a model to predict whether a movie is a "Horror" or "Comedy" from the description and title. The total dataset has 506,244 records. First, using TensorFlow, we trained a LSTM-based model to predict these categories. The first layer of this model computes what is called a word-embedding, where the LSTM learns a feature-space in which similar words (co-occuring) are closer together. The next two layers consist of dense layers that map the words from the feature-space to classification outputs. The result is a model that achieves 93% accuracy, which is far more accurate than simpler alternatives on a Bag-of-Words featurization (random forests 90%, Linear SVM 81%, Kernel SVM 85%).

**Fraud:** ProPublica collected a dataset of corporate donations to medical researchers to analyze conflicts of interest. Records contain the PI's medical specialty, the drug brand name (null if not drug), the device brand name (null if not a device), name of pharamceutical donor, the amount donated, and whether the research is disputed. The dataset comes with a `status` field that describes whether or not the donation was allowed under the declared research protocol. We used a Multi-Layer Perceptron to classify disallowed donations which achieved a 82% accuracy (random forest 81%, Linear SVM 80%, Kernel SVM 80%).

### 4.3 Experiments

**Exp 1. Isolating Mispredictions** In the first experiment, we illustrate one benefit of PALM in terms of how it can explain predictions in terms of subsets of training data. We measure the quality of an explanation in the following way: (1) select a test data point that is mispredicted, (2) construct an explanation of this prediction by selecting a subset of training data, (3) for all examples outside of the selected subset randomly flip the labels of 25% of the points, (4) retrain the model and observe if the test data point changes its prediction. We run 1000 trials of this procedure for 10 randomly sampled mispredictions and plot the results in Figure 2. This metric measures how well isolated the selected neighborhood is from the other points–in other words, if the other points were changed how much would the prediction change. The random baseline changes it prediction roughly 25% of the time. The clustering and the nearest neighbor approaches only consider the feature-space, not the structure of the classifier. While they are significantly more robust than random ($\leq 5\%$) prediction changes, PALM is much more robust ($\leq 1$ %) prediction changes.

**Exp 2. Discovering other Misprediction with PALM** In the next experiment, we explore whether mispredictions concentrate around particular submodels. We train the models on 80% of the dataset, and test on the remaining 20%. We measure the fraction of mispredictions attributed to each submodel. We want to show that mispredictions concentrate around specific submodels and are not evenly distributed throughout the feature-space. Figure 3 shows this effect when we apply our algorithm to approximate the Movie and Fraud models with $k = 10$ submodels. For the Movie dataset, 70% of the mis-predictions are attributed to a single submodel. For the Fraud dataset, 78% of the errors are attributed to two of the submodels.

[1] ftp://ftp.fu-berlin.de/pub/misc/movies/database/
[2] http://webscope.sandbox.yahoo.com/catalog.php?datatype=r

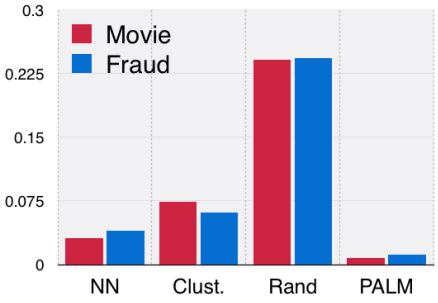


**Figure 2: PALM isolates relevant training data more effectively than baselines.**

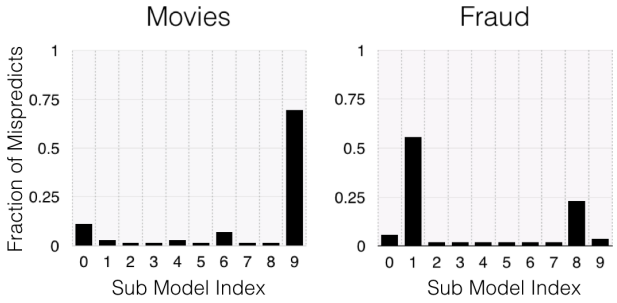


**Figure 3: On two datasets, we show how mispredictions concentrate around specific submodels. This means there are specific regions of the feature-space most associated with mispredictions.**

**Exp 3. Agreement With the Original Model** We now measure the agreement between the PALM model with the original model as we increasing the number of partitions $k$ (Figure 4, where agreement is the accuracy with respect to the original model. We find that with $k = 16$, both PALM models still retain $> 90\%$ agreement with the original model. This suggests that PALM can help developers identify more precise subsets of the training data while still trusting that the PALM models are reflective of the original model.

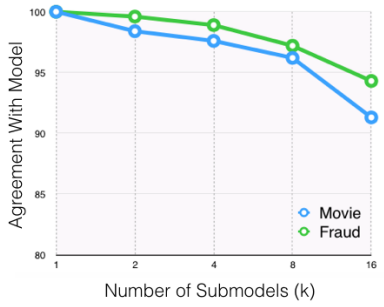


**Figure 4: On two datasets, we find that the discretized models agree with greater than 90% accuracy with the original model. As the number of submodels increase the accuracy goes down.**

**Exp 4. PALM is efficient** PALM can run offline during training time and queries to PALM are very efficient as it simply involves a decision tree evaluation. PALM is explicitly designed to mimic the user's model, and the decision tree learned by the meta-model can easily be used to index or partition the training data can be indexed. For the nearest neighbors approach, it is possible to quickly find the neighbors by using intelligent indexing structures such as KD-trees or Oct-trees. However spatial indices are well known to have difficulty scaling to very high dimensional datasets. In addition, finding the neighbors in a high dimensional space may simply not identify relevant results. One could apply dimensionality reduction but this would further reduce its efficacy. On the other hand the clustering approach is much faster because it only requires querying each cluster center. Our interesting insight is that PALM is much faster than the nearest neighbor approach, and competitive with the clustering approach in terms of run time. Our approach returns more than 30x faster than a nearest neighbor search even when coupled with dimensionality reduction and a KD-Tree. The clustering approach is much faster but PALM is still within the latencies needed for interactive latencies.

**Table 1: Run times of the different algorithms in seconds**

| Algorithm | Movie | Fraud |
|---|---|---|
| kNN | 35.61 | 22.11 |
| kNN+PCA | 12.34 | 6.97 |
| PALM | 0.334 | 0.31 |
| Clustering | 0.07 | 0.06 |

## 5 RECENT WORK AND NEXT STEPS

The study of model intepretability and explainability is recently a hot topic in ML research [13, 14, 16], especially in the context of Neural Networks. An example of one such approach is to train a sparse linear model in the local neighborhood of a point[13]. Another more recent approach in computer vision is to use attention models, enforce that the model focuses on certain features, for explainability [9]. Another relevant line of work is Neural Network Rule Extraction [7]. This problem is very challenging since highly expressive deep models such as deep neural networks, which in principle can learn any deterministic function, are susceptible to adversarial examples (i.e., imperceptible perturbations to the features that cause a change in prediction)[15]. Explaining a prediction exactly in terms of features is highly useful for an end-user, but developers also need to be able to trace modeling errors to training data [12]. This is why our approach focuses on identifying the most informative neighborhood (partition) of training data. We believe that techniques that isolate features and data are complementary and hope to explore combinations of the two in the future.

Based on our initial study and survey of recent related work, we have highlighted a number of important challenges for the future systems.

**Connecting Explainability to Data Provenance:** We believe that there is further work to be done to explain predictions in terms of relevant source data. Systems like PALM can be connected to lineage systems to trace even further upstream than just the training data. This leads a number of computational challenges in storing, processing, and summarizing the selected tuples.

**Reducing Hyper-Parameters and Failure Modes:** Ironically, existing work in explainable models, including PALM, all have subtle failure modes due to their assumptions and hyper-parameters. This could lead to faulty or misleading explanations and erode user trust. We hope to explore techniques that require less tuning and less detailed understanding of the mathematical structure of the problem in the future.

**Scalability of Human Effort:** Finally, an important concern is how human analysts can explore and iterate through large training datasets. While systems like PALM can reduce the burden, there are still many more hurdles before truly useful machine learning debuggers. We believe that coupling explanations with anomaly detection may be a viable next step, as in the MacroBase project [3].

**Acknowledgements: This research was supported in part by a seed grant from the UC Berkeley Center for Information Technology in the Interest of Society (CITRIS), the UC Berkeley RISELab, and by the U.S. National Science Foundation under Award IIS-1227536: Multilateral Manipulation by Human-Robot Collaborative Systems. This work has been supported in part by funding from Google and and Cisco.**

## REFERENCES

[1] *Keystone ML*. http://keystone-ml.org/.
[2] *Tensor Flow*. https://www.tensorflow.org/.
[3] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. 2017. Macrobase: Prioritizing attention in fast data. SIGMOD.
[4] James Cheney, Laura Chiticariu, Wang-Chiew Tan, and others. 2009. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases* 1, 4 (2009), 379–474.
[5] Andrew Crotty, Alex Galakatos, and Tim Kraska. 2014. Tupleware: Distributed Machine Learning on Small Clusters. In *IEEE Data Eng. Bull.* http://sites.computer.org/debull/A14sept/p63.pdf
[6] Roy Fox*, Sanjay Krishnan*, Ken Goldberg, and Ion Stoica. 2017. Multi-Layer Deep Option Discovery. In *Under Review ICML*.
[7] Tameru Hailesilassie. 2016. Rule Extraction Algorithm for Deep Neural Networks: A Review. *arXiv preprint arXiv:1610.05267* (2016).
[8] Joseph M. Hellerstein, Christopher Re, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. In *VLDB*. http://vldb.org/pvldb/vol5/p1700_joehellerstein_vldb2012.pdf
[9] Jinkyu Kim and John Canny. 2017. Interpretable Learning for Self-Driving Cars by Visualizing Causal Attention. *arXiv preprint arXiv:1703.10631* (2017).
[10] Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. 2013. MLbase: A Distributed Machine-learning System. In *CIDR*. http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper118.pdf
[11] Sanjay Krishnan, Animesh Garg, Richard Liaw, Lauren Miller, Florian T. Pokorny, and Ken Goldberg. 2016. HIRL: Hierarchical Inverse Reinforcement Learning for Long-Horizon Tasks with Delayed Rewards. *CoRR* abs/1604.06508 (2016). http://arxiv.org/abs/1604.06508
[12] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. 2016. ActiveClean: Interactive Data Cleaning For Statistical Modeling. *PVLDB* 9, 12 (2016), 948–959. http://www.vldb.org/pvldb/vol9/p948-krishnan.pdf
[13] Tao Lei, Regina Barzilay, and Tommi Jaakkola. 2016. Rationalizing neural predictions. *arXiv preprint arXiv:1606.04155* (2016).
[14] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Why Should I Trust You?: Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1135–1144.
[15] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).
[16] Jessica Taylor, Eliezer Yudkowsky, Patrick LaVictoire, and Andrew Critch. 2016. *Alignment for advanced machine learning systems*. Technical Report. Technical Report 20161, MIRI.
[17] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. In *VLDB*. http://www.vldb.org/pvldb/vol6/p553-wu.pdf