Mash: software tools for developing interactive and transparent machine learning systems

An T. Nguyen
University of Texas at Austin
atn@cs.utexas.edu

Matthew Lease
University of Texas at Austin
ml@utexas.edu

Byron C. Wallace
Northeastern University
byron@ccs.neu.edu

ABSTRACT

We present *Mash*, a library that provides software tools for developing interactive and transparent machine learning systems. Mash provides tools for (1) back end: specifying machine learning models, and (2) front end: generating User Interfaces (UIs) that allow non-technical end-users to interpret and interact with the models. In the back-end, Mash includes probabilistic modeling and computation graph utilities. On the front-end, UIs generated by Mash enable end users to examine the model's predictions, manipulate model's parameters, and observe changes in model's predictions. Together, Mash can help developers rapidly build new models and obtain feedback from end users. We present two case studies in which Mash UIs enable end-users to: (1) assess model fairness in predicting student performance, and (2) explore movie recommendations. To support future work, we will open source Mash.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Human centered computing** → *User interface toolkits*; • **Software and its engineering** → *Development frameworks and environments.*

KEYWORDS

Machine Learning; Explainable; Software tool.

ACM Reference Format:
An T. Nguyen, Matthew Lease, and Byron C. Wallace. 2019. Mash: software tools for developing interactive and transparent machine learning systems. In *Joint Proceedings of the ACM IUI 2019 Workshops (IUI Workshops'19)*, 7 pages.

1 INTRODUCTION

The traditional goal of machine learning (ML) is to create accurate predictors. In a typical development process, ML developers are provided a dataset and are tasked with building a prediction system that achieves high performance as measured by standard metrics such as accuracy. The system is then often presented to end users as a black box, with little interactive capability. A user interface (UI) may be built, but the interactions are often limited to specifying the inputs and observing the outputs; the black box remains opaque. Although that development process has been successful, there are many applications where richer end-user interactions are beneficial or even necessary: when the systems are used to make high-stakes decisions that affect people, when human knowledge or creativity is needed, or when the capability to explore the systems' predictions may help engage users.

Developing ML systems capable of providing rich interactions is a difficult and lengthy process, as it involves the integration of back-end predictive model building with front-end UI design. If developers focus only on optimizing predictive performance without considering end users then the system may become too complex for end users to meaningfully interact with.

To address these issues, we propose Mash, a library for developing interactive and transparent ML systems. For developers, Mash facilitates 'user-aware' ML development, where predictive models are integrated with associated UIs. For end-users, Mash-generated UIs enable rich interactions with ML systems, helping the users make decisions, explore the systems, or inject their knowledge into the system.

Advances in ML are often accompanied by software libraries. The success of Support Vector Machines (SVMs) in the 2000s was due in part to software packages such as LIBSVM [11] and SVMLight [21]. Recent progress in ML has been accelerated by the availability of many probabilistic programming and computation graph libraries (PyMC [28], Stan [9], Theano [5], Tensorflow [1], PyTorch [27], and others). The creation of specialized software tools for transparent and interactive ML, such as Mash, has the potential to similarly accelerate progress.

Mash integrates existing ML libraries with tools for generating UIs automatically, including tools for specifying how an end-user can interacts with the models (for example, the end-user can inspect and manipulate some of the model parameters and observe how these change the prediction).

IUI Workshops'19, March 20, 2019, Los Angeles, USA

Given these tools, developers can build systems with end-users in mind: they can construct or identify intelligible components in the systems to expose to end users. For instance, developers may define a model with a sparsity constraint (requiring a large number of parameters to be zeros) and define the UI to include only non-zero parameters for users to interact with.

We demonstrate the applicability of Mash in two case studies. First, we consider predicting student performance using linear regression, mixed effects model, and neural networks. Such predictions can be used to make important decisions, which may raise the question of fairness. For example, a ML system that supports student admission decisions [36] must be fair with respect to race and gender. Second, we present an interactive movie recommendation UI powered by probabilistic matrix factorization [31], allowing users to directly manipulate model parameters to explore recommendations.

2 RELATED WORK

Model visualization

Software tools for visualizing ML models exist: TensorBoard [1], ShinyStan [9], ModelTracker [4] and others [19]. However, they aim at helping developers debug models, or helping expert users (who have ML knowledge) understand and deploy systems. Tools that aim at non-expert users (Tensorflow Playground or ConvnetJS) are mainly educational and are used in toy problems. In contrast, we aim at helping developers generate UIs to assist non-expert end-users in interpreting and interacting with ML models in real world problems.

The recently released What-if tool[1] enables the manipulation of the data to observe how the prediction changes. Mash goes further in enabling the manipulation of the *internal* model parameters. Furthermore, What-if is a post-hoc tool on built models, while Mash is a development tool for building models.

Interpretable machine learning

Two main directions in this area are developing inherently interpretable models, and (2) probing black box models to generate explainations. In the first direction, popular methods are decision trees [29], sparse linear models [33], and additive models [10]). Although these methods facilitate user interpretation, their predictive performance can be limited.

In the second direction, recent work has considered using a simpler model to approximate a complex black-box model [30] or finding the most influential training examples [22]) for each test instance. These methods can provide insights into black-box models. However, their explanations may not be faithful since they are simpler approximations to complex models.

There has also been concern that 'interpretable' is not a well-defined concept [24], and that a more 'rigorous science' is needed [14]. Other than providing the tools for developers, we see Mash as a framework toward a 'science of ML interpretation'. By building the general tools instead of focusing on a model or an application, we expect to see general principles for ML interpretation.

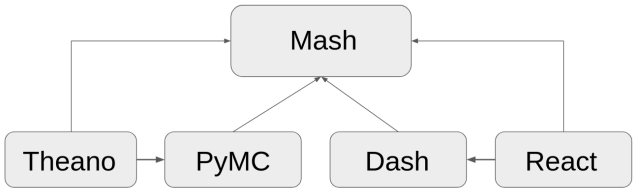

Figure 1: Mash and its main dependencies.

Interactive machine learning:

Work in this area aims at using human interaction to improve ML systems [3] in a number of applications: image segmentation [15], text classification [23], and image search [16]. Our work aims at creating the tools for building UIs that enable the interaction between humans and ML systems. We focus on system transparency, although user interaction can be used to improve system prediction performance.

Development tools for human computation

Turkit [25] is a tool for developing iterative crowdsourcing tasks, such as iteratively improving text [6]. Jabberwocky [2] provides a programming environment for social computing. These tools have roughly the same goal: providing an abstraction layer connecting back-end and front-end components in order to help developers focus on high-level ideas in human computation algorithms. Mash has a similar goal for developing ML systems.

3 BACKGROUND

Computation graphs

Computation graph libraries (Theano [5], Tensorflow [1], PyTorch [27] and others) provide the tools for defining operators (from scalars, vectors, and matrices to tensors) and operations (arithmetic, indexing, conditioning). The key functionality of these libraries is automatic differentiation, which enables gradient based parameter learning. In the development of new deep neural networks models, these computation graph libraries have become essential: without them, developers would need to tediously derive gradients for all variables in the networks (and repeat after every change).

[1] https://ai.googleblog.com/2018/09/the-what-if-tool-code-free-probing-of.html

Probabilistic programming

Probabilistic programming systems (PyMC [28], Stan [9], BUGS [32], Anglican [34] and others) provide the tools for defining probabilistic models, which are probability distributions over a collection of random variables. Probabilistic models can also be interpreted as computation graphs in which the operators (scalars, vectors, ...) are 'lifted' from fixed values to random variables. Similar to automatic differentiation, probabilistic programming systems provide the key functionality of automatic inference: infering the distribution over all variables given the observed data. Markov Chain Monte Carlo [17] and Variational Inference [35] are two families of commonly implemented automatic inference methods.

Probabilistic modeling is the foundation for many popular ML algorithms: mixture models, probabilistic matrix factorization for recommendation [31], Latent Dirichlet Allocation (LDA) topic models [7], and others. The advantage of a probabilistic approach is the clear meaning of variables in the models and the uniform representation of uncertainty. However, this approach often does not scale well to very large datasets.

4 SYSTEM

Mash uses Theano [5] for computation graphs, PyMC [28] for probabilistic modeling, Dash[2] for web application building, and React[3] for user interface rendering. **Figure 1** shows Mash and its dependencies on other tools.

Consider a linear regression model to predict student grades from two features: 'study time' and 'health' (we will later use this example as a case study). Linear regression simply assumes that the target (grade) is a linear combination of the features:

$$\text{grade} = \alpha + \beta_S \times \text{studytime} + \beta_H \times \text{health} + \epsilon \qquad (1)$$

where α, β_S, and β_H are the parameters to be learned and ϵ is Normally distributed random noise. Taking a Bayesian approach, we place zero-mean Normal priors on all parameters. These priors help regularize parameter estimates to avoid over fitting.

Source code for using Mash to implement this model and specifying the UI is shown below:

```
import mash as ms
m = ms.Model()
# Assume data is a Pandas dataframe with 3 columns:
# studytime, health, and observed_grade
m.set_pd_data(data)

with m:
    # Priors on parameters
    alpha = ms.Normal('alpha', mu=0, sd=10)
    beta_S = ms.Normal('beta_S', mu=0, sd=10)
    beta_H = ms.Normal('beta_H', mu=0, sd=10)
    sigma = ms.HalfNormal('sigma', sd=1)

    # Likelihood
    grade = ms.Deterministic('Grade', alpha +
        beta_S * m.data['studytime'] +
        beta_H * m.data['health'])
    observed_grade = ms.Normal('Observed_Grade',
        mu=grade, sd=sigma,
        observed=m.data['observed_grade'])

# Set the roles of variables
m.set_prediction(features=['studytime', 'health'],
                 params=[alpha, beta_S, beta_H],
                 predict=grade)

m.inference()

# Define UI
ui = ms.UI(model=m)
ui.add_sliders(['studytime', 'health',
    'alpha', 'beta_S', 'beta_H', 'Grade'])
ui.run_server()
```

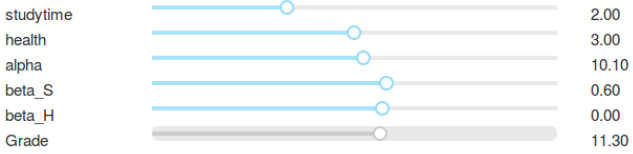

Figure 2: UI generated by Mash for Linear Regression (with Bayesian priors) for predicting student grades.

Figure 2 shows the generated web UI, which consists of six sliders. The two data sliders (studytime and health) are set to their default values (they can also be set to the values of some test datapoints). The three model parameter sliders (alpha, beta_S, and beta_H) are set to the values that the model has learned from the training data. These first five sliders can be manipulated by the end-users, while the prediction slider (Grade) reacts to these manipulations. For example, if an end-user moves the 'studytime' slider, the 'Grade' slider will move to the model's predicted grade for a student with the selected study time. End-users can also move a model parameter slider such as alpha and observe the change in the model's prediction. This interaction may help end-users make sense of how the model works internally. The overall UI also provides transparency in letting end-users see and interact with model's parameters. We note that the developers make the decisions on which data features and model parameters to add to the UI. There is a trade-off between adding all parameters for more transparency vs. adding fewer parameters for easier end-user sense-making.

[2]https://github.com/plotly/dash/
[3]https://github.com/facebook/react

In general, developers use Mash by declaring an ML model and declaring the corresponding UI. Under the hood, Mash creates a PyMC probabilistic model, extracts a Theano computation graph, matches the UI elements to features or model parameters, sets up callbacks for handling interactions, and renders the interface.

For back-end ML model building, Mash provides:

- Placeholders for data.
- Common probability distribution (e.g. Normal).
- Operations on data or model parameters (arithmetic, indexing, and matrix operations).

For front-end UI, Mash provides:

- Sliders for manipulating data or parameters, or for observing model's predictions.
- Graphs for plotting data.
- Markdown text for explaining the model and instructing end-users.

Mash also connects back-end and front-end components automatically, enabling developers to focus on declaring what they want in the ML model and the UI.

5 CASE STUDIES

Predicting student performance

Some institutions have used ML systems to support making student admission decisions to their programs [36], due to the rapid increase in the number of applicants. Although reducing the workload for the admission committee, these systems raise the issue of fairness and transparency: how do we know these systems do not discriminate between students based on race or gender, even though these attributes are not used as features for the predictors? For example, a common feature is the student's current school. If a system has a negative weight on an all-female school then that system may discriminate between students based on gender (although it is also possible that the system has correctly identified a school with low quality teaching). Fairness in ML is an active research area, such as in the context of credit rating or criminal justice. Many criteria for fairness have been proposed [26], such as equality of opportunity [18] or demographic parity [8], but there is still no consensus. Technical criteria may provide some insight, but fairness is a more complex concept, which necessitates human interpretation.

In this case study, we address the issue by making the system transparent to the end-users, who might be responsible for making admissions decisions. We use the Student Performance Dataset from the UCI repository [12, 13] and implement three models: linear regression, a mixed effects model, and neural networks.

Our linear regression implementation is presented in the previous section. In this case study, to help end-users assess model fairness, we include an interactive histogram of the model's predictions with respect to student gender (**Figure 3**):

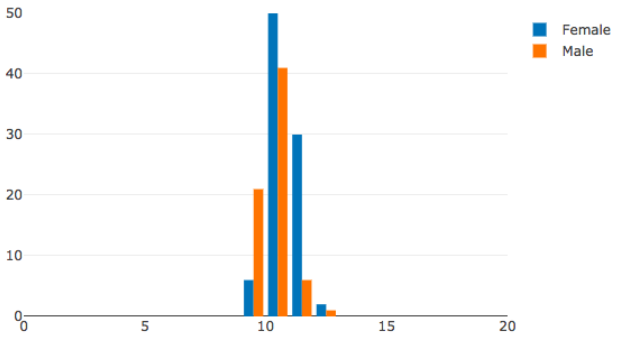

Figure 3: The interactive histogram showing the model's predicted grades by student gender: the x-axis is the grade, the y-axis is the number of students in the provided test dataset.

```
m.set_pd_test_data(test_data)
ui.add_test_histogram(hue='gender')
```

The interactive histogram reacts to end-users' manipulations of the model parameters. If the end-users determine that there are biases, they can try to manipulate model parameters to reduce these biases. Other visualization techniques for assessing fairness are possible. For example, another histogram may show the distribution of the prediction errors for each gender.

Our mixed effects model extends linear regression in specifying a random effect on the intercept parameter α based on the school that the student attends[4]:

```
1  school_alphas = ms.Normal('school_alphas',
2      mu = alpha, sd=1, shape=(2,))
3  grade = ms.Deterministic('Grade',
4      school_alphas[m.data['school']] +
5      beta_S * m.data['studytime'] +
6      beta_H * m.data['health'])
```

In the dataset, there are two schools called MS and GP. The first line generated an intercept α for each school. Next, in line 4, we select the intercept based on the school that the student attends (`m.data['school']` is 0 if the school is MS and 1 if the school is GP). **Figure 4** shows our interface, in which end-users can specify the school of the student (in addition to study time and health). End-users can also inspect and manipulate the intercept α for each of the two schools.

For (Bayesian) neural networks, we implement a model with two hidden units, F0 and F1. The model's parameters are: weights from the inputs to the hidden units, and weights from the hidden units to the final grade prediction. The value of a hidden unit is a linear combination of the inputs passed through a non-linear activation function (we use the tanh

[4]This random effect can be written in R formula as (alpha|school)

Figure 4: The UI for the mixed effects model.

Figure 5: The interface for our Bayesian neural networks.

function):

$$\text{F0} = \tanh[\text{studytime} \times W(\text{studytime}, \text{F0}) + \text{health} \times W(\text{health}, \text{F0})] \quad (2)$$

Where W(X,Y) is the weight from X to Y (for example the weight W(health, F0) connects the input health to the first hidden unit F0). In **Figure 5**, we display our UI. End-users can manipulate data, manipulate model parameters, and observe the predicted grade (similar to previous UIs). A potential issue is that the hidden units F0 and F1 have no obvious interpretations. Furthermore, for applications with a large number of features, the large number of parameters in a neural network may make it difficult for user to inspect and manipulate. To address these, developers can implement sparsity or disentangled representations [20] to reduce the number of parameters and construct hidden units with clearer meanings.

Movie recommendation

Recommendation systems provide suggestions of items to users. For instance, a movie recommendation system may suggest new movies to a user given the ratings that user gave to other movies, often leveraging a large dataset of previous ratings.

Explanations for recommendation systems exist and have been widely used in commercial systems such as in Netflix, but they are typically static, such as 'people like you also watch'. In this case study, we consider making a recommendation system transparent and interactive to users. Users can directly manipulate the model's parameters, and observe how the recommendation changes. This is useful for users to see how the system works, explore different recommendations, or find recommendations for friends.

One of the most common techniques for recommendation is Probabilistic Matrix Factorization (PMF) [31], in which the system identifies and estimates a number of hidden factors for each item (movie) and user (viewer). Assuming that there are n users and m movies. The idea in PMF is to factor the $n \times m$ ratings matrix into two matrices: a $n \times k$ user matrix U and a $m \times k$ movie matrix M, where k is the number of factors (which is usually set to a small number, we set $k = 3$). PMF assumes that the rating by user i for movie j is the dot product of two k dimension row vectors: U_i and M_j. Intuitively, U_i describes user i and M_j describes movie j. Our implementation first defines the PMF model:

```
# User matrix
Users = ms.Normal('Users', mu=0, sd=1,
    shape=(n_users, n_factors))

# Movie matrix
Movies = ms.Normal('Movies', mu=0, sd=1,
    shape=(n_movies, n_factors))

# Rating matrix
rating = ms.Deterministic('Rating',
Users[m.data['userId']]
.dot(Movies[m.data['movieId']].T))
```

We next define the UI:

```
# Sliders for selecting user and movie
userId = ui.add_slider('userId')
movieId = ui.add_slider('movieId')

# Shows the name of the selected movie
ui.add_dynamic_text(data=movie_names,
    from_index=movieId)

# Sliders for user factors
for i in range(n_factors):
    ui.add_slider('Users', (0, i),
        display_name='User F' + str(i))

# Sliders for movie factors
for i in range(n_factors):
    ui.add_slider('Movies', (0, i),
        display_name='Movie F' + str(i))

# The predicted rating
ui.add_slider('Rating')
```

In **Figure 6**, we show our UI, where end-users can: (1) select a user (movie viewer) and a movie, (2) manipulate the

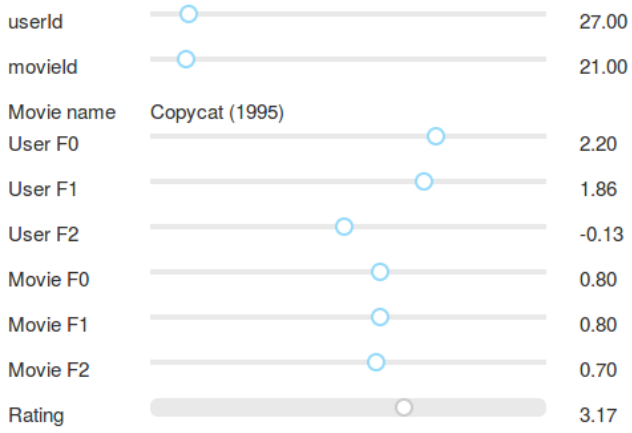

Figure 6: The interface for our movie recommendation system based on PMF.

factor values for the selected user and movie, and (3) observe the predicted rating (also for the selected user and movie). A potential difficulty is that the factors (F0, F1, and F2) have no clear meanings. To better understand the predictions, the end-users may need to look at a large number of movies and movie factors. For example, if the value of 'Movie F0' is consistently high for action movies, then the factor F0 may be correlated with that genre. End-users can then interpret the value of 'User F0' as a measure of how much the selected user likes action movies.

6 DISCUSSION

Conclusion

We have presented Mash, a software library for developing interactive and transparent ML systems. We also presented our Mash implementations of four models in two case studies. We see Mash as a step toward better interactions between human and ML systems.

Limitations and future work

Mash is built on top of PyMC [28], which has a limitation in scaling to large datasets, and Theano [5], which is no longer in active development. Future work could integrate Mash to more recent ML libraries such as Tensorflow [1] or PyTorch [27]. The main interactions in Mash-built UIs are manipulating sliders and observing model predictions, which could be difficult for end-users in applications with a large number of features or parameters. Future work may develop techniques for interacting with high-dimensional features or parameters.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their time in reviewing this paper. This work is supported in part by National Science Foundation grant No. 1253413. Any opinions, findings, and conclusions or recommendations expressed by the authors are entirely their own and do not represent those of the sponsoring agencies.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI*.

[2] Salman Ahmad, Alexis Battle, Zahan Malkani, and Sepandar D. Kamvar. 2011. The jabberwocky programming environment for structured social computing. In *UIST*.

[3] Saleema Amershi, Maya Cakmak, W. Bradley Knox, and Todd Kulesza. 2014. Power to the People: The Role of Humans in Interactive Machine Learning. *AI Magazine* 35 (2014), 105–120.

[4] Saleema Amershi, David Maxwell Chickering, Steven M. Drucker, Bongshin Lee, Patrice Y. Simard, and Jina Suh. 2015. ModelTracker: Redesigning Performance Analysis Tools for Machine Learning. In *CHI*.

[5] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf*, Vol. 1.

[6] Michael S. Bernstein, Greg Little, Rob Miller, Björn Hartmann, Mark S. Ackerman, David R. Karger, David Crowell, and Katrina Panovich. 2010. Soylent: a word processor with a crowd inside. In *UIST*.

[7] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[8] Toon Calders, Faisal Kamiran, and Mykola Pechenizkiy. 2009. Building classifiers with independency constraints. In *Data mining workshops, 2009. ICDMW'09. IEEE international conference on*. IEEE, 13–18.

[9] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of statistical software* 76, 1 (2017).

[10] Rich Caruana, Yinjun Lou, Johannes Gehrke, Paul Koch, Marc Sturm, and Noémie Elhadad. 2015. Intelligible Models for HealthCare: Predicting Pneumonia Risk and Hospital 30-day Readmission. In *KDD*.

[11] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)* 2, 3 (2011), 27.

[12] Paulo Cortez and Alice Maria Gonçalves Silva. 2008. Using data mining to predict secondary school student performance. (2008).

[13] Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. (2017). http://archive.ics.uci.edu/ml

[14] Finale Doshi-Velez and Been Kim. 2017. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608* (2017).

[15] Jerry Alan Fails and Dan R. Olsen. 2003. Interactive machine learning. In *IUI '03*.

[16] James Fogarty, Desney S. Tan, Ashish Kapoor, and Simon A. J. Winder. 2008. CueFlik: interactive concept learning in image search. In *CHI*.

[17] Stuart Geman and Donald Geman. 1984. Stochastic relaxation, Gibbs

distributions, and the Bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence* 6 (1984), 721–741.

[18] Moritz Hardt, Eric Price, and Nathan Srebro. 2016. Equality of Opportunity in Supervised Learning. In *NIPS*.

[19] Fred Hohman, Minsuk Kahng, Robert Pienta, and Duen Horng Chau. 2018. Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers. *IEEE transactions on visualization and computer graphics* (2018).

[20] Sarthak Jain, Edward Banner, Jan-Willem van de Meent, Iain James Marshall, and Byron C. Wallace. 2018. Learning Disentangled Representations of Texts with Application to Biomedical Abstracts. In *EMNLP*.

[21] Thorsten Joachims. 1999. Svmlight: Support vector machine. *SVM-Light Support Vector Machine http://svmlight. joachims. org/, University of Dortmund* 19, 4 (1999).

[22] Pang Wei Koh and Percy Liang. 2017. Understanding Black-box Predictions via Influence Functions. In *ICML*.

[23] Todd Kulesza, Margaret M. Burnett, Weng-Keen Wong, and Simone Stumpf. 2015. Principles of Explanatory Debugging to Personalize Interactive Machine Learning. In *IUI*.

[24] Zachary Chase Lipton. 2018. The mythos of model interpretability. In *CACM*.

[25] Greg Little, Lydia B. Chilton, Max Goldman, and Rob Miller. 2010. TurKit: human computation algorithms on mechanical turk. In *UIST*.

[26] Arvind Narayanan. 2018. FAT* tutorial: 21 fairness definitions and their politics. *ACM Conference on Fairness, Accountability, and. Transparency* (2018).

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[28] Anand Patil, David Huard, and Christopher J Fonnesbeck. 2010. PyMC: Bayesian stochastic modelling in Python. *Journal of statistical software* 35, 4 (2010), 1.

[29] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.

[30] 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.

[31] Ruslan Salakhutdinov and Andriy Mnih. 2007. Probabilistic Matrix Factorization. In *NIPS*.

[32] David Spiegelhalter, Andrew Thomas, Nicky Best, and Wally Gilks. 1996. BUGS 0.5: Bayesian inference using Gibbs sampling manual (version ii). *MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK* (1996), 1–59.

[33] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), 267–288.

[34] David Tolpin, Jan-Willem van de Meent, and Frank Wood. 2015. Probabilistic programming in Anglican. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 308–311.

[35] Martin J Wainwright, Michael I Jordan, and others. 2008. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning* 1, 1–2 (2008), 1–305.

[36] Austin Waters and Risto Miikkulainen. 2014. GRADE: Machine Learning Support for Graduate Admissions. *AI Magazine* (2014). http://nn.cs.utexas.edu/?waters:aimag14