Threshy: Supporting safe usage of intelligent web services

Alex Cummaudo
ca@deakin.edu.au
Applied Artificial Intel. Inst.
Deakin University
Geelong, Australia

Scott Barnett
scott.barnett@deakin.edu.au
Applied Artificial Intel. Inst.
Deakin University
Geelong, Australia

Rajesh Vasa
rajesh.vasa@deakin.edu.au
Applied Artificial Intel. Inst.
Deakin University
Geelong, Australia

John Grundy
john.grundy@monash.edu
Faculty of Inf. Tech.
Monash University
Clayton, Australia

ABSTRACT

Increased popularity of ‘intelligent’ web services provides end-users with machine-learnt functionality at little effort to developers. However, these services require a decision threshold to be set which is dependent on problem-specific data. Developers lack a systematic approach for evaluating intelligent services and existing evaluation tools are predominantly targeted at data scientists for pre-development evaluation. This paper presents a workflow and supporting tool, Threshy, to help *software developers* select a decision threshold suited to their problem domain. Unlike existing tools, Threshy is designed to operate in multiple workflows including pre-development, pre-release, and support. Threshy is designed for tuning the confidence scores returned by intelligent web services and does not deal with hyper-parameter optimisation used in ML models. Additionally, it considers the financial impacts of false positives. Threshold configuration files exported by Threshy can be integrated into client applications and monitoring infrastructure. Demo: https://bit.ly/2YKeYhE.

CCS CONCEPTS

• **Information systems → Web services**; • **Computing methodologies → Artificial intelligence**; • **Software and its engineering → Designing software**; *Software post-development issues.*

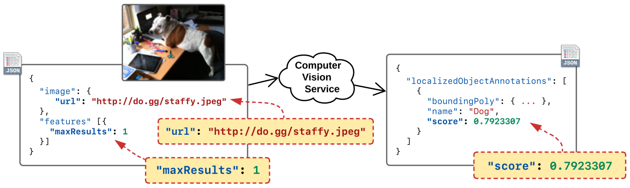
KEYWORDS

intelligent services, tooling, thresholding, decision theory

ACM Reference Format:
Alex Cummaudo, Scott Barnett, Rajesh Vasa, and John Grundy. 2020. Threshy: Supporting safe usage of intelligent web services. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3368089.3417919

1 INTRODUCTION

Machine learning (ML) algorithm adoption is increasing in modern software. End users routinely benefit from machine-learnt functionality through personalised recommendations [4], voice-user interfaces [12], and intelligent digital assistants [3]. The easy accessibility and availability of intelligent web services[1] is contributing to their adoption. These intelligent web services simplify the development of ML solutions as they (i) do not require specialised ML expertise to build and maintain AI-based solutions, (ii) abstract away infrastructure related issues associated with ML [2, 14], and (iii) provide web APIs for ease of integration.



```
{
  "image": {
    "url": "http://do.gg/staffy.jpeg"
  },
  "features" [{
   "maxResults": 1
  }]
}
```

```
"url": "http://do.gg/staffy.jpeg"
```

```
"maxResults": 1
```

```
{
  "localizedObjectAnnotations": [
    {
      "boundingPoly": { ... },
      "name": "Dog",
      "score": 0.7923307
    }
  ]
}
```

```
"score": 0.7923307
```

Figure 1: Request and response for an intelligent computer vision web service with only three configuration parameters: the image's `url`, `maxResults` and `score`.

However, unlike traditional web services, the functionality of these *intelligent services* is dependent on a set of assumptions unique to ML [6]. These assumptions are based on the data used to train ML algorithms, the choice of algorithm, and the choice of data processing steps—most of which are not documented. For developers, these assumptions mean that the performance characteristics of an intelligent service in any particular application problem domain is not fully knowable. Intelligent services represent this uncertainty through a confidence value associated with their predictions.

As an example, consider fig. 1, which illustrates an image of a dog uploaded to a real computer vision service. Developers have very few configuration parameters in the upload payload (`url` of the image to analyse and `maxResults` the number of objects to detect). The JSON output payload returns the confidence value via a `score` field (0.792), the bounding box and a “dog” label. Developers can only work with these parameters; unlike hyper-parameter optimisation available to ML creators, who can configure the internal parameters of the algorithm while training a model. Given the structure of the abstractions, developers have no insight into which hyper-parameters are used or the algorithm selected and cannot tune the underlying trained model when using an intelligent service. Thus an evaluation procedure must be followed as a part of using an intelligent service for an application to work with and tune the output confidence values for a given input set.

[1] Such as Azure Computer Vision (https://azure.microsoft.com/en-au/services/cognitive-services/computer-vision/), Google Cloud Vision (https://cloud.google.com/vision/), or Amazon Rekognition (https://aws.amazon.com/rekognition/).

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417919

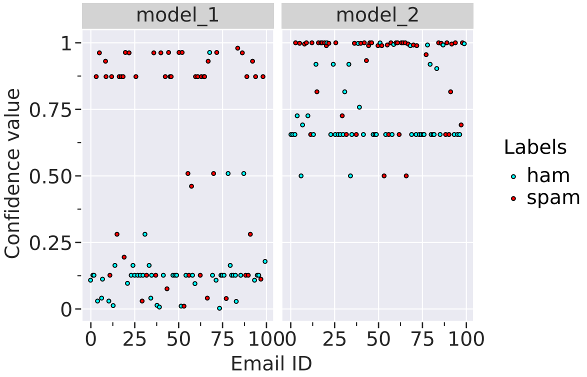
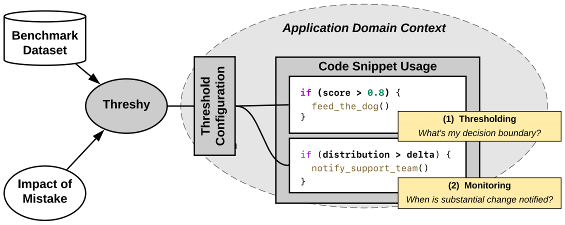


Figure 2: Predictions for 100 emails from two spam classifiers. Decision thresholds are classifier-dependent: a single threshold for both classifiers is *not* appropriate as ham emails are clustered at 0.12 (model_1) and at 0.65 (model_2). Developers must evaluate performance for *both* thresholds.

A typical evaluation process involves a test data set (curated by the developers using the intelligent service) that is used to determine an appropriate threshold. Choice of a decision threshold is a critical element of the evaluation procedure [9]. This is especially true for classification problems such as detecting if an image contains cancer. Simple approaches to selecting a threshold are often insufficient, as highlighted in Google's ML course: *"It is tempting to assume that [a] classification threshold should always be 0.5, but* ***thresholds are problem-dependent, and are therefore values that you must tune****."*[2]

As an example consider the predictions from two email spam classifiers shown in Figure 2. The predicted safe emails, 'ham', are in two separate clusters (a simple threshold set to approx. 0.2 for model 1 and 0.65 for model 2, indicating that different decision thresholds may be required depending on the classifier. Also note that some emails have been misclassified; how many depends on the choice of decision threshold. An appropriate threshold considers factors outside algorithmic performance, such as financial cost and impact of wrong decisions. To select an appropriate decision threshold, developers using intelligent services need approaches to reason about and consider trade-offs between competing *cost factors*. These include impact, financial costs, and maintenance implications. Without considering these trade-offs, sub-optimal decision thresholds will be selected.

The standard approach for tuning thresholds in classification problems involve making trade-offs between the number of false positives and false negatives using the receiver operating characteristic (ROC) curve. However, developers (i) need to realise that this trade-off between false positives and false negatives is a data dependent optimisation process [15], (ii) often need to develop custom scripts and follow a trial-and-error based approach to determine a threshold, (iii) must have appropriate statistical training and expertise, and (iv) be aware that multi-label classification require more complex optimisation methods when setting label specific costs. However, current intelligent services do not sufficiently guide or support software engineers through the evaluation process, nor do they make this need clear in the documentation.



```
if (score > 0.8) {
  feed_the_dog()
}
```

```
if (distribution > delta) {
  notify_support_team()
}
```

Figure 3: Threshy supports two key aspects for intelligent web services: threshold selection and monitoring.

In this paper we present ***Threshy***[3], a tool to assist developers in selecting decision thresholds when using intelligent services. The motivation for developing Threshy arose from our work across a set of industry projects, and is an implemented example of the threshold tuner component presented in our complementing ESEC/FSE 2020 architecture tactic publication [5]. While Threshy has been designed to specifically handle pre-trained classification ML models where the hyperparameters cannot be tuned, the overall conceptual design serves as inspiration for general model calibration. Unlike existing tooling (see section 4), **Threshy serves as a means to up-skill and educate software engineers in selecting machine-learnt decision thresholds**, for example, on aspects such as confusion matrices. We re-iterate that the end-users of Threshy are software engineers and not data scientists—Threshy is not designed for hyper-parameter tuning of models, but for threshold tuning to use intelligent web services more robustly where internal models are not exposed. Threshy provides a visually interactive interface for developers to fine-tune thresholds and explore trade-offs of prediction hits/misses. This exposes the need for optimisation of thresholds, which is dependent on particular use cases.

Threshy improves developer productivity through automation of the threshold selection process by leveraging an optimisation algorithm to propose thresholds. Figure 3 illustrates the two key aspects by which Threshy supports developer's application domain context. Developers input a representative dataset of their application data (a benchmark dataset) in addition to cost factors to Threshy. Threshy's output helps developers select appropriate thresholds while considering different cost factors and can be used to monitor the evolution of an intelligent service. Developers also benefit from the workflow implemented in Threshy by providing a reproducible procedure for testing and tuning thresholds for any category of classification problem (binary, multi-class, and multi-label). The output, is a configuration file that can be integrated into client applications ensuring that the thresholds can be updated without code changes, and continuously monitored in a production setting.

2 MOTIVATING EXAMPLE

As a motivating example consider Nina, a fictitious developer, who has been employed by Lucy's Tomato Farm to automate the picking of tomatoes from their vines (when ripe) using computer vision

[2] See https://bit.ly/36oMgWb.

[3] Threshy is available for use at http://bit.ly/a2i2-threshy

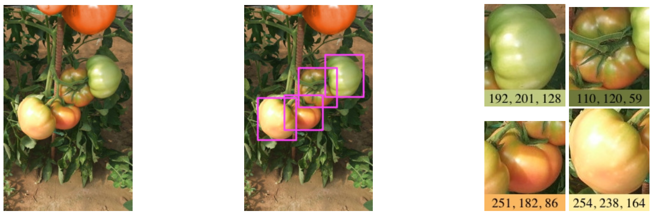


Figure 4: Pipeline of Nina's harvesting robot. *Left:* Photo from harvesting robot's webcam. *Centre:* Classification detecting different types of tomatoes. *Right:* Binary classification for ripeness (ripe/unripe) based on (R, G, B values).

and a harvesting robot. Lucy's Farm grow five types of tomatoes (roma, cherry, plum, green, and yellow tomatoes). Nina's robot—using an attached camera—will crawl and take a photo of each vine to assess it for harvesting. Nina's automated harvester needs to sort picked tomatoes into a respective container, and thus several business rules need to be encoded into the prediction logic to sort each tomato detected based on its *ripeness* (ripe or not ripe) and *type of tomato* (as above). Nina uses a two-stage pipeline consisting of a multi-class and a binary classification model. She has decided to evaluate the viability of cloud based intelligent services and use them if operationally effective. Figure 4 illustrates the pipeline used:

(1) **Classify tomato 'type'.** This stage uses an object localisation service to detect all tomato-like objects in the frame and classifies each tomato into one of the following labels: [`'roma'`,`'cherry'`,`'plum'`,`'green'`,`'yellow'`,`'unknown'`].
(2) **Assess tomato 'ripeness'.** This stage uses a crop of the localised tomatoes from the original frame to assess the crop's colour properties (i.e., average colour must have R > 200 and G < 240). This produces a binary classification to deduce whether the tomato is ripe or not.

Nina only has a minimal appreciation of the evaluation method to use for off-the-shelf computer vision (classification) services. She also needs to consider the financial costs of misclassifying either the tomato type or the ripeness. Missing a few ripe tomatoes isn't a significant concern as the robot travels the field twice a week during harvest season. However, picking an unripe tomato is expensive as Lucy cannot sell them. Therefore, Nina needs a better (automated) way to assess the performance of the service and set optimal thresholds for her picking robot, to maximise profit.

To assist in developing Nina's pipeline, Lucy sampled a section of 1000 tomatoes by taking a photo of each tomato, manually labelling its type, and assessing whether the vine was `'ripe'` or `'not_ripe'`. Nina ran the labelled images through an intelligent service, with each image having a predicted type (multi-class) and ripeness (binary), with respective confidence values.

Nina combined the predictions, their respective confidence values, and Lucy's labelled ground truths into a CSV file which was then uploaded to Threshy. Nina asked Lucy, the farmer, to assist in setting relevant costs (from a business perspective) for correct predictions and false predictions. Threshy then recommended a choice of decision threshold which Nina then fine tuned while considering the performance and cost implications.

3 THRESHY

Threshy is a tool to assist software engineers with setting decision thresholds when integrating machine-learnt components in a system in collaboration with subject matter experts. Our tool also serves as a method to inform and educate engineers about the nuances to consider when using prepackaged ML services. Key novel features are:

- Automating threshold selection using an optimisation algorithm (NSGA-II [7]), optimising the results for each label.
- Support for user defined, domain-specific weights when optimising thresholds, such as financial costs and impact to society. This allows decision thresholds to be set within a business context as they differ between applications [8].
- Handles nuances of classification problems such as dealing with multi-objective optimisation, and metric selection—reducing errors of omission.
- Support key classification problems including binary (e.g. email is spam or ham), multi-class (e.g. predict the colour of a car), and multi-label (e.g. assign multiple topics to a document). Existing tools ignore multi-label classification.

Setting thresholds in Threshy is an eight step process as outlined in fig. 5. Software engineers ① run a benchmark dataset through the machine-learnt component to create a data file (CSV format) with true labels and predicted labels along with the predicted confidence values. The data file is then ② uploaded for initial exploration where engineers can ③ experiment with modifying a single global threshold for the dataset. Developers may choose to exit at this point (as indicated by dotted arrows in fig. 5). Optionally, the engineer ④ defines costs for missed predictions followed by selecting optimisation settings. The optional optimisation step of Threshy ⑤ considers the performance and costs when deriving the thresholds. Finally, the engineer can ⑥ review and fine tune the calculated thresholds, associated costs, and ⑦ download generated threshold meta-data to be ⑧ integrated into their application.

Threshy runs a client/server architecture with a thin-client (see fig. 6). The web-based application consists of an interactive front-end where developers upload benchmark results—consisting of both human annotated labels and machine predictions from the intelligent service—and use threshold tuners (via sliders) to present a data summary of the uploaded information. Predicted model performances and costs are entered manually into the web interface by the developer. The Threshy back-end runs a data analyser, cost processor and metrics calculator when relevant changes are made to the front-end's tuning sliders.

The data analyser provides a comprehensive overview of confusion matrices compatible for multi-label multi-class classification problems. When representing the confusion matrix, it is trivial to represent instances where multi-label multi-classification is not considered. However, a more challenging case to visualise arises when you have n labels and m classes as the true/false matches become too excessive to visualise; $n * m * 4$ fields need to be presented. We resolve this challenge by summarising the statistics down to three constructs: (i) number of true positives, (ii) false positives, and (iii) missed positives. This allows us to optimise against the true positives and minimise the other two constructs.

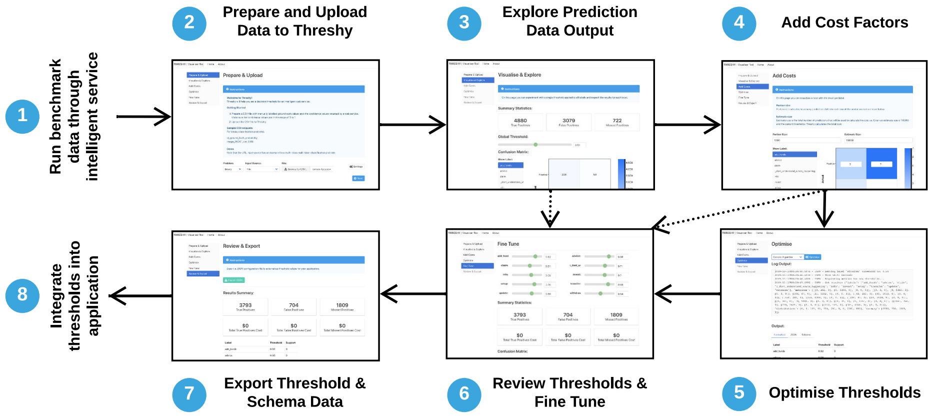

1 Run benchmark data through intelligent service
2 Prepare and Upload Data to Threshy
3 Explore Prediction Data Output
4 Add Cost Factors
5 Optimise Thresholds
6 Review Thresholds & Fine Tune
7 Export Threshold & Schema Data
8 Integrate thresholds into application

Figure 5: UI workflow for interacting with Threshy to optimise the thresholds for classification problem.

Threshy is a fully self-contained repository of the tool implementation, scripting and exploratory notebooks, which is available at https://github.com/a2i2/threshy.

4 RELATED WORK

Optimal machine-learnt decision boundaries depend on identifying the operating conditions of the problem domain. A systematic study by Drummond and Holte [8] classifies four operating conditions to determine a decision threshold: (i) the operating condition is known and the model trained matches perfectly; (ii) where the operating conditions are known but change with time, and thus the model must be adaptable to such changes; (iii) where there is uncertainty in the knowledge of the operating conditions certain changes in the operating condition are more likely than others; and (iv) where there is no knowledge of the operating conditions and the conditions may change from the model in any possible way. Various approaches to determine appropriate thresholds exist for all four of these cases, such as cost-sensitive learning, ROC analysis, and Brier scores. However, an *automated* attempt to calibrate decision threshold boundaries is not considered, and is largely pitched at a non-software engineering audience. A recent study touches on this in model management for large-scale adversarial instances in Google's advertising system [15], however this is only a single component within the entire architecture, and is not a tool that is useful for developers in varying contexts. Threshy provides a 'plug-and-play' style calibration method where any context/domain can have thresholds automatically calibrated *and* optimised for engineers. Threshy's architecture supports a headless mode for use in monitoring workflows.

Support tools for ML frameworks generally fall into two categories. The first attempts to illuminate the 'black box' by offering ways in which developers can better understand the internals of the model to improve its performance. For extensive analyses and surveys into this area, see [11, 13]. However, a recent emphasis to probe only inputs and outputs of a model has been explored, exploring off-the-shelf models without knowledge of its unknowns (see fig. 2) to reflect the nature of real-world development. Google's *What-If Tool* [16] for Tensorflow provides a means for data scientists to visualise, measure and assess model performance and fairness with various hypothetical scenarios and data features; similarly, Microsoft's *Gamut* tool [10] provides an interface to test hypotheticals on Generalized Additive Models, and a *ModelTracker* tool [1] collates summary statistics on sample data to enable visualisation of model behaviour and access to key performance metrics.

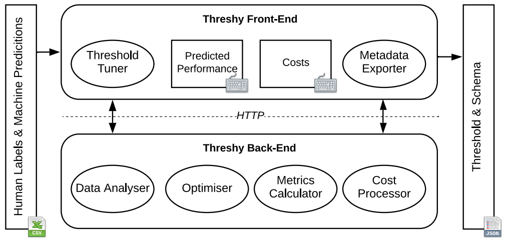


Figure 6: Architecture of Threshy.

However, these tools are focused toward pre-development model evaluation and not designed for software engineering workflows. Nor are they context-aware to the overall software system they are meant to target. They are also aimed at data scientists and model builders and do not consider consistent tooling that works across development, test, and production environments. They also do not provide synthesised output for using intelligent web services with predetermined thresholds. . Further, certain tools are tied to specific ML frameworks (e.g., What-If and Tensorflow). Our work, instead, attempts to bridge these gaps through a context-aware, structured workflow with an automated tool targeted to software developers; our tool is designed for software engineers to calibrate thresholds and is used for intelligent service APIs in particular.

5 CONCLUSIONS & FUTURE WORK

Primary contributions of this work include Threshy, a tool for automating threshold selection, and the overall meta-workflow proposed in Threshy that developers can use as a point of reference for calibrating thresholds. Threshy only deals with classification problems and adapting our method to other problem domains is left as future work. Furthermore, we plan to evaluate Threshy with practitioners for user-acceptance and add support for code synthesis for calibrating the API responses.

REFERENCES

[1] Saleema Amershi, Max Chickering, Steven M Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. 2015. Modeltracker: Redesigning performance analysis tools for machine learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, Seoul, Republic of Korea, 337–346. https://doi.org/10.1145/2702123.2702509

[2] Anders Arpteg, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. 2018. Software engineering challenges of deep learning. In *Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, Prague, Czech Republic, 50–59. https://doi.org/10.1109/SEAA.2018.00018

[3] Matt Boyd and Nick Wilson. 2018. Just ask Siri? A pilot study comparing smartphone digital assistants and laptop Google searches for smoking cessation advice. *PLoS ONE* 13, 3 (2018). https://doi.org/10.1371/journal.pone.0194811

[4] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, Boston, MA, USA, 191–198. https://doi.org/10.1145/2959100.2959190

[5] Alex Cummaudo, Scott Barnett, Rajesh Vasa, John Grundy, and Mohamed Abdelrazek. 2020. Beware the evolving ‘intelligent’ web service! An integration architecture tactic to guard AI-first components. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Sacramento, CA, USA. https://doi.org/10.1145/3368089.3409688 In Press.

[6] Alex Cummaudo, Rajesh Vasa, John Grundy, Mohamed Abdelrazek, and Andrew Cain. 2019. Losing Confidence in Quality: Unspoken Evolution of Computer Vision Services. In *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution*. IEEE, Cleveland, OH, USA, 333–342. https://doi.org/10.1109/ICSME.2019.00051

[7] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (April 2002), 182–197. https://doi.org/10.1109/4235.996017

[8] Chris Drummond and Robert C Holte. 2006. Cost curves: An improved method for visualizing classifier performance. *Machine Learning* 65, 1 (October 2006), 95–130. https://doi.org/10.1007/s10994-006-8199-5

[9] Moritz Hardt, Eric Price, and Nathan Srebro. 2016. Equality of opportunity in supervised learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Curran Associates Inc., Barcelona, Spain, 3323–3331. https://doi.org/978-1-51-083881-9

[10] Fred Hohman, Andrew Head, Rich Caruana, Robert DeLine, and Steven M Drucker. 2019. Gamut: A design probe to understand how data scientists understand machine learning models. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, Glasgow, Scotland, UK. https://doi.org/10.1145/3290605.3300809

[11] Fred Hohman, Minsuk Kahng, Robert Pienta, and Duen Horng Chau. 2019. Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers. *IEEE Transactions on Visualization and Computer Graphics* 25, 8 (2019), 2674–2693. https://doi.org/10.1109/TVCG.2018.2843369

[12] Chelsea Myers, Anushay Furqan, Jessica Nebolsky, Karina Caro, and Jichen Zhu. 2018. Patterns for how users overcome obstacles in Voice User Interfaces. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, Vol. 2018-April. ACM, Montreal, QC, Canada, 6. https://doi.org/10.1145/3173574.3173580

[13] Kayur Patel, James Fogarty, James A Landay, and Beverly Harrison. 2008. Investigating statistical machine learning as a tool for software development. In *Proceedings of the 26th SIGCHI Conference on Human Factors in Computing Systems (CHI ’08)*. ACM, Florence, Italy, 667–676. https://doi.org/10.1145/1357054.1357160

[14] D Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean François Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*. Curran Associates Inc., Montreal, QC, Canada, 2503–2511. https://doi.org/10.5555/2969442.2969519

[15] D Sculley, Matthew Eric Otey, Michael Pohl, Bridget Spitznagel, John Hainsworth, and Yunkai Zhou. 2011. Detecting adversarial advertisements in the wild. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, San Diego, CA, USA, 274–282. https://doi.org/10.1145/2020408.2020455

[16] James Wexler, Mahima Pushkarna, Tolga Bolukbasi, Martin Wattenberg, Fernanda Viegas, and Jimbo Wilson. 2019. The What-If Tool: Interactive Probing of Machine Learning Models. *IEEE Transactions on Visualization and Computer Graphics* 26, 1 (2019), 56–65. https://doi.org/10.1109/tvcg.2019.2934619 arXiv:1907.04135