Evolving Software to be ML-Driven Utilizing Real-World A/B Testing: Experiences, Insights, Challenges

Paul Luo Li, Xiaoyu Chai, Frederick Campbell, Jilong Liao, Neeraja Abburu, Minsuk Kang,
Irina Niculescu, Greg Brake, Siddharth Patil, James Dooley, and Brandon Paddock
Microsoft
Redmond, USA
(pal,xicha,frcampbe,jilli,neabbu,mika,irnicule,gbrake,spatiil,jdooley,bpaddock)@microsoft.com

Abstract—**ML-driven software is heralded as the next major advancement in software engineering; existing software today can benefit from being evolved to be ML-driven. In this paper, we contribute practical knowledge about evolving software to be ML-driven, utilizing real-world A/B testing. We draw on experiences evolving two software features from the Windows operating system to be ML-driven, with more than ten real-world A/B tests on millions of PCs over more than two years. We discuss practical reasons for using A/B testing to engineer ML-driven software, insights for success, as well as on-going real-world challenges. This knowledge may help practitioners, as well as help direct future research and innovations.**

Index Terms—**Software engineering, data analysis, software quality, software development management, learning (artificial intelligence), machine learning, machine learning algorithms, predictive models, big data applications**

I. Introduction

Machine learning (ML)-driven software is heralded as the next major advancement in computing: Software 2.0 [1]. Traditionally, software developers manually program rules and heuristics to determine the behavior of software; for ML-driven software, developers program a skeleton of the machine learning model, provide it with examples, and then let the machine "learn" the optimal behavior. In this manner, software can leverage the vast amount of data being collected today to determine best behaviors and to continuously improve user experience. ML-driven software is becoming increasingly prevalent today with reported uses in PCs [2], phones [3], IoT devices [4], and automobiles [5].

Much attention has been given to software like facial recognition [6], voice recognition [7], autonomous driving [8], among others built from the ground-up to be ML-driven; nonetheless, existing software today can also benefit from ML [2]. Little is known today about evolving existing software to be ML-driven, especially using A/B testing, which is the gold-standard used by many software organizations today to design, evaluate, and deploy innovations [9] [10]. In this paper, **we contribute practical knowledge about evolving software to be ML-driven utilizing real-world large-scale A/B testing**:

- Practical reasons for using A/B testing to build ML-driven software.
- Insights for successfully evolving software to be ML-driven using A/B testing.
- Real-world challenges for evolving ML-driven software using A/B testing.

We report on evolving two software features from the Windows operating system to be ML-driven: Windows Update (WU) and System Initiated User Feedback (SIUF). The design, evaluation, deployment, and update of these ML-driven software features involved more than ten real-world A/B tests on millions of PCs over more than two years. Our experiences can help and inspire practitioners to evolve their software to be ML-driven; the challenges and issues we faced can help direct research and future innovations.

For the rest of this paper, we first discuss related work and background in Sections II and III. The methodology is discussed in Section IV. We then describe the two real-world software features that we evolved to be ML-driven in Section V. We discuss four reasons for using A/B testing to build ML-driven software in Section VI. The insights of successfully evolving software to be ML-driven using A/B testing are discussed in Section VII. We discuss three on-going challenges in Section VIII. Finally, we conclude in Section IX.

II. Related Work

In this section, we discuss related work on A/B testing and ML-driven software.

A. A/B Testing

In its simplest form, an A/B test randomly assigns users to one of two variants: control (A) or treatment (B). Usually control is the existing system and treatment is the software with a feature added/changed, say, feature X. User interactions with the two software variants are recorded and from that, metrics are computed and compared. If the A/B test is designed and executed correctly, the only difference between the two variants is feature X. External factors such as seasonality, impact of other feature changes, competitor moves, etc. are distributed evenly between control and treatment, and therefore do not impact results of the experiment. Hence, differences in metrics between the two groups (in aggregate) can be attributed to either feature X or noise. The noise part is quantified with statistical tests (e.g. t-test). The upshot is that a causal relationship between the change to the product and changes in user behaviors is established through A/B testing. Experimentation has been shown to be especially useful when user reactions are uncertain to design changes or novel innovations [11], which provides organizations an effective approach of making data-driven decisions on their innovations.

Due to its growing importance in the software industry, A/B testing is an active research area. Research has been focused on topics such as new statistical methods to improve metric sensitivity [12], metric design and interpretation [13], projections of results from a short-term A/B test to the long term [14], the benefits of A/B testing at scale [11], A/B testing in a social network setting [15], high-level architecture of A/B testing platforms [16], [17], privacy and fairness concerns [18], as well as examples, pitfalls, rules of thumb and lessons learned from running controlled experiments in practical settings [19].

Some research papers have discussed developing ML using "experiments", which entails trying different approaches to build the ML (e.g. configurations and algorithms [20]). In this paper, we refer A/B testing to be randomized controlled trials in real-world conditions with actual users, and we examine the utilization of A/B testing in building ML-driven software.

B. Machine Learning Driven Software

At its core, machine learning instructs computing devices to use example/past data to solve a given problem. It uses statistical models to encode the example/past data in order to make inferences about the future [21]. For example, in supervised learning, computers are presented with example inputs (typically contextual factors associated with the occurrence) and desired outputs. The goal is to learn rules–a model–that map the inputs to the outputs. Then when provided new inputs, the computing devices can use the model to make an inference.

With increasing computing capabilities of edge devices and growth in data volumes being collected, machine learning has been receiving growing interest in all areas of computing and has been investigated for many applications. Examples include mobile keyboard word completion [3], automotive traffic predictions [5], image identification [22], and even medical prognostications [23]. Research on machine learning has seen matching growth, with efforts proceeding on many fronts, including new algorithms [24], new domains of applications [25], ethical/fairness considerations [26] [27], adversarial machine learning [28], architecture/design of real-world systems [29], as well as experience reports and how-to-guides for real-world practitioners [30].

Much attention in both research and industry has been devoted to novel software applications built from the ground-up to be ML-driven, including facial recognition [6], voice recognition [7], autonomous driving [8]. However, little has been reported about evolving the vast amount of existing software today to be ML-driven, even though existing software can also benefit from ML [2]. Evolving existing software to be ML-driven is the focus of this paper.

III. Background

In this section, we provide background of ML infrastructure and A/B testing capabilities on the Windows operating system.

A. Windows Intelligent Services Engine

The ML-driven software features discussed in this paper are built/evolved using the Windows Intelligent Services Engine (WISE): a Windows internal platform for ML-driven software features on Windows PCs [31]. WISE is an end-to-end solution, with components that link together cloud-side ML training (via Azure Machine Learning) and client-side inferences (e.g. WinML). It abstracts away the complexities of building, deploying, A/B testing, and maintaining ML models from Windows engineering teams. Once deployed, the ML model can be continuously evolved/improved, in a data-driven manner, without the need to update application codes. The system (patent pending) has four integrated parts: cloud-compute ML training/retraining pipeline, scalable and secure deployment infrastructure, configurable client-side data manager and prediction orchestrator, as well as a mechanism to execute A/B testing.

For an application developer using WISE, the first step is to register their scenario with the cloud ML training/retraining pipeline and receive a unique identifier. The scenario is the phenomenon of interest that the ML model would predict, which the application can then use to modify its behavior. The developer provides example/past data of the phenomenon to train an initial model. One of the key features of WISE is providing a default set of common predictors on Windows devices (discussed more in Section V), which enables developers of different applications to reuse in their individual ML models. The application developer builds their intelligent application by calling a client-side API with unique identifiers. The call returns an inference about the phenomenon of interest specific to the context of the device (e.g. time during the day), which the application can then use to adjust behaviors. Behind the API, WISE securely deploys models to devices using a cloud-service. In addition to the model (e.g. an ONNX file), a feature engineering file is also deployed, which specifies what raw data to use and how to transform them to produce the predictors. Since the model and feature configurations are behind the API call, the ML model can be updated on the cloud without changing the application codes. The API also records telemetries from users who have opted to share their data for product improvement, which can be sent back to the cloud for model improvement. Iterations of the ML model can be evaluated and rolled out using a randomized controlled trials methodology with the Windows Experimentation Platform (WExp) [17], in order to monitor and evaluate causal impact of the ML model.

B. The Windows Experimentation Platform

WISE evaluates the causal impact of ML models with the Windows Experimentation Platform (WExp): a client-side A/B testing platform for the Windows operating system [17]. WExp leverages several existing components/services at Microsoft, while building new components and using existing ones in innovative ways. For example, the delivery of assets (e.g. codes and ML models) to devices leverages the Windows Update service [32]. Data are recorded and transmitted using Windows Telemetry services and adheres to user settings and restrictions [33]. Randomization and statistical comparisons of data are performed using the ExP system [16]. Finally, WExp has a set of policies and processes that help to ensure safety and best-practices. For example, all A/B tests intended to reach the general population must first be ran on internal and/or Windows Insiders devices (a self-selected group of users who run pre-release versions of Windows operating system and provide feedback [34]) to ensure safety and quality.

WExp enables validation, deployment, and update of ML-driven features in several ways. During feature development, WExp can A/B test the ML-driven feature (on or off) as well as A/B/C test various model configurations (e.g. triggering thresholds). After feature is deployed, WExp supports reverse A/B tests to further validate the feature efficacy in production, as well as controlled rollouts of updates (i.e. rollouts of new models with randomized control groups for causal inferences).

IV. Methodology

Our case studies of evolving existing software to be ML-driven using A/B testing cover work in the Windows organization at Microsoft corporation in the USA between 2018 and 2020. The authors of this paper are or have been employed at Microsoft, who work on the WISE platform and/or ML-driven software as software developers or data scientists. We briefly describe our empirical approach as prescribed by Runeson and Höst [35].

A. Data Collection

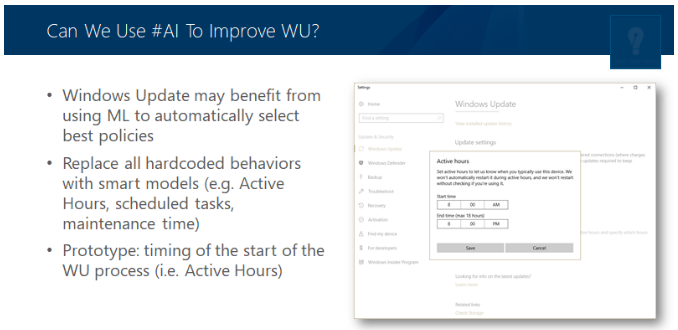
We leveraged two primary sources of data. First, we used materials from actual A/B tests, including analysis reports produced by engineering teams, which the authors had access to. These contained findings, conclusions, and subsequent engineering decisions. Second, we used documentation (e.g. notes, emails, presentations, and design docs) extracted from document depots (e.g. Sharepoint).

B. Data Analysis

Overall, we present factual findings from building and A/B testing of the ML-driven software, including efforts and decisions in response to the A/B test results. Our analyses and insights are based on the authors' knowledge and experiences.

C. Threats to Validity

To reduce the risk to constructing validity, we name software features, components, and findings (where possible). We use industry standard terminologies and provide explanations/examples to clarify ambiguous concepts and terms. We reference public communications, patents, and external reports where possible. To reduce threats to internal validity,


Can We Use #AI To Improve WU?

- Windows Update may benefit from using ML to automatically select best policies
- Replace all hardcoded behaviors with smart models (e.g. Active Hours, scheduled tasks, maintenance time)
- Prototype: timing of the start of the WU process (i.e. Active Hours)

Windows Update

Active hours

Set active hours to let us know when you typically use this device. We won't automatically restart it during active hours, and we won't restart without checking if you're using it.

Start time: 8 00 AM

End time (max 18 hours): 8 00 PM

Fig. 1. Initiative framing slide for ML-driven Windows Update effort, showing an example of a heuristic that can be improved using ML.

this paper has been reviewed internally to ensure accuracy. Finally, though this paper is a case-study with external validity limitations, many of the topics and concerns are not tied directly to the Windows operating system and are applicable to other contexts and organizations.

V. ML-Driven Windows Features

In this section, we describe two Windows features and their efforts to enhance behaviors with ML. The two engineering teams self-selected to pioneer evolving their Windows features to be ML-driven using WISE.

A. Windows Update

Windows Update (WU) provides the latest ”patches” to the files and applications on Windows PCs [36]. Many of the updates are of vital importance, such as those involving security and/or application functionality. Consequently, PCs are commonly configured to automatically install updates and reboot to ensure that the updates take effect. The install and reboot processes ensure that updates are installed in a timely manner, whiling using rules and heuristics to minimize user disruptions, such as outside of ”Active Hours”, as shown in Figure 1. However, user feedback indicated that the rules and heuristics could be improved: an anecdote being a user actively working on their PC outside of active hours, stepping away momentarily, and then returning to find their PC in the middle of a reboot.

The engineering team leveraged WISE to enhanced WU behaviors with ML. When attempting a potentially disruptive reboot post-installation, a functionality was added to predict whether the PC will be active using ML, and then possibly adjusting behaviors (e.g. defers the reboot). This was referred internally as Smart Busy Check (SBC). In addition to reducing user disruptions, the engineering team also believed that leveraging ML could improve upgrade velocity. Since users often choose to defer (i.e. not complete updates) when notified of impending reboots, the ML model can also improve upgrade velocity by avoiding potentially disruptive situations and reducing reboot deferrals. These motivations are shown in the original initiative framing slide in Figure 1.

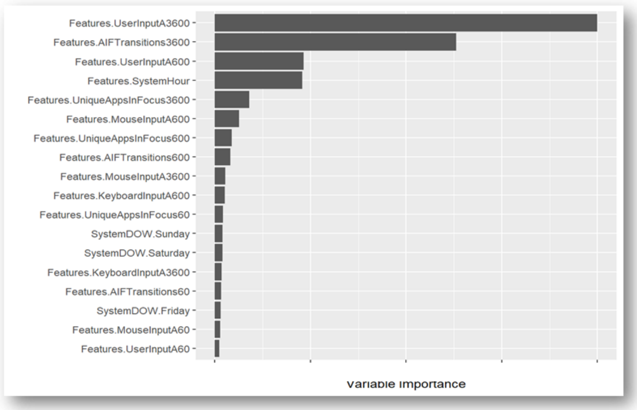


Fig. 2. Feature importance plot for the WU ML model, showing the diversity of contributing predictors.

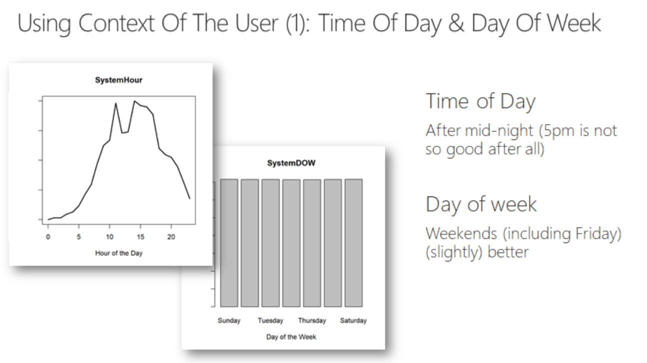


Fig. 3. Insights about environmental factors for the WU ML model, showing how time of day (e.g. after mid-night) and day of week impact outcomes.

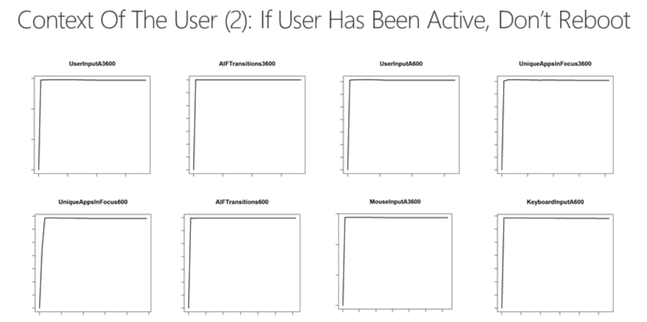


Fig. 4. Insights about user interactions for the WU ML model, showing how any recent user interactions impact outcomes.

The WU ML model uses predictors on PC client that consist of environmental factors like time of day and day of week, as well as user interactions with the device (e.g. mouse or keyboard activities). The model is gradient boosted trees, implemented using Microsoft's LightGBM algorithm. This algorithm was chosen due to contextual considerations and the nature of tabular telemetry data (discussed more in Section VII-D). The model was constructed under standard practices (e.g. train/validation/test splits and parameter tuning via random cross validations). The initial model was trained using data from Windows Insiders, with 51 predictors and 5.5M observations from 2.5K devices over 3 months. The model was then updated using production data from general users who opted to share their data for product improvement [33]. Figure 2 shows the feature importance plot of the ML model. Insights about the impact of environmental and user interaction features are in Figures 3 and Figure 4. For example, data show that users are frequently active at 5pm (the default for Active Hours) and that it may not be an optimal time for rebooting. An interesting note is that a same underlying feature can be engineered into different predictors that capture different aspects. For example, the predictors include the amount of activities in the past 60, 600, and 3600 seconds. When considered together in a same model, it identifies the most relevant duration to examine as well as the interplay between these features.

In addition to feature/functional/automated testing, the WU ML-driven software feature underwent three phases of A/B testing. First, like many other Windows features, this ML-driven software feature initially underwent ON/OFF testing on the internal self-host population to verify feature quality. These A/B tests help to uncover unforeseen issues, such as cross-component system integration issues. The primary focus of these A/B tests is quality rather than efficacy; these initial A/B tests are ran on populations different from the intended audience and in contexts that are different from the intended environment.

Second, the team used A/B tests on Windows Insiders to further validate quality and assess model configurations (e.g. probability thresholds). Though success of the feature cannot be fully determined on Beta users in a pre-release environment (discussed more in Section VIII), these A/B tests do assess possible negative reactions to the software feature. As a result, the software feature was shipped after quality validation was completed and initial ML configurations were set.

Released in the Windows 10 1909 update, the ML-driven WU received positive feedback from users and reviewers [2]. The team monitored the performance of this ML-driven software feature as the 1909 Update was progressively deployed. Once the deployed population was sufficiently large and representative of the overall population, the team used production data to update the ML model. This is an underappreciated aspect of ML-driven applications for edge devices, which is that edge devices (especially phones and PCs) do not all update instantaneously or consistently (i.e. some devices update faster than others). Therefore, there are usually biases present in the initial data returned, and determining when to update an ML model can sometimes be difficult. After the ML model was updated, the engineering team again went through multiple rounds of A/B tests to validate efficacy. The engineering team then used controlled feature rollouts–the third kind of A/B testing–to progressively release the update while monitoring quality (using a control group). Today, it continues the monitoring and updating of this ML-driven

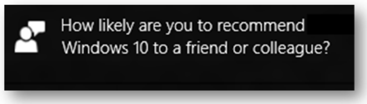

How likely are you to recommend Windows 10 to a friend or colleague?

Fig. 5. Screenshot of a Windows SIUF question to be improved using ML.

software feature.

B. *System Initiated User Feedback*

System Initiated User Feedback (SIUF) ”surveys” selected users about their experiences with Windows PCs [37]. SIUF gives valuable feedback/inputs to engineering teams by providing “in the moment” feedback from users after they’ve performed certain actions. It uses both rating scales and free form texts to facilitate easy evaluations, and also enables asking granular/targeted questions about specific aspects of the users’ experiences. Figure 5 shows an example SIUF about the overall customer experiences with Windows PCs (usually appearing on the lower right of the screen). SIUF questions are triggered by conditions (e.g. number of usage hours and/or usage of specific Windows features) and are managed by various rules (e.g. asking each question only once and not asking more than one question in a period of time). Feedback and statistics on the SIUF system suggested that improvements can be made to survey users when they are more likely to respond and less likely to be disrupted.

The engineering team also leveraged WISE to enhance SIUF behaviors with ML. In addition to meeting existing conditions and rules, SIUF was enhanced with an option to use an ML model that predicts whether the user is likely to respond. The model was a lightGBM model built using the same standard practices as the WU ML model. The SIUF is then only asked if the likelihood of response is high (otherwise deferred to a future time), making better use of the chance to survey the user (possible bias/fairness concerns are discussed in Section VIII-A). The SIUF ML model uses environmental factors (e.g. day of week), user interactions with the device (e.g. mouse and keyboard activities), as well as device characteristics (e.g. country and locale). The starting model was trained using data from a production SIUF. Figure 6 displays the feature importance plot, and the insights about key predictors are shown in Figure 7 and Figure 8.

Similar to WU, the ML-driven SIUF went through multiple phases of A/B testing. These included multiple iterations of A/B tests to ensure quality, as well as verifying no negative user reactions. While quality being high, the A/B tests uncovered various contextual issues that lead to interesting results about user reactions (discussed more in Section VIII). Nevertheless, since the ML-driven functionality was built as an *addition to existing*, the engineering team was able to ship it and continue A/B testing without impacting other SIUF questions in production.

VI. Reasons for using A/B testing to build ML-driven software

In this section, we discuss four reasons for using A/B testing to build ML-driven software based on our experiences: safety

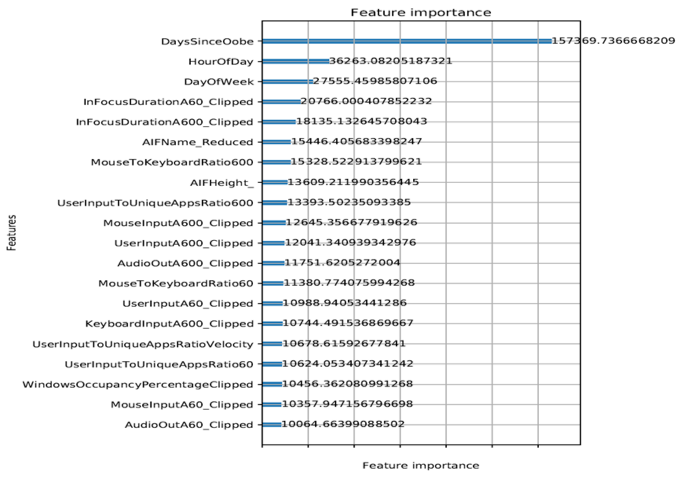


Fig. 6. Feature importance plot for the SIUF ML model, showing the diverse contributing predictors.

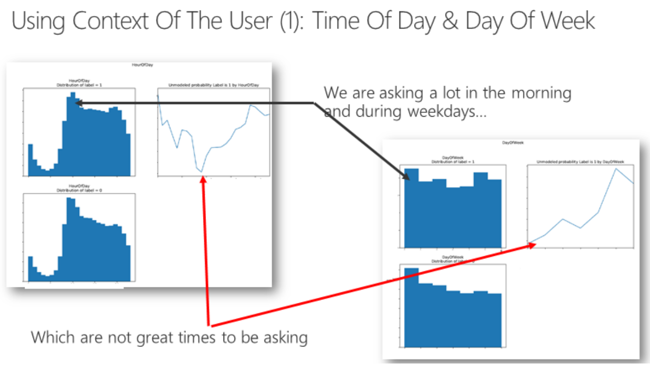


Fig. 7. Insights about environmental factors for the SIUF ML model, showing how time of day (e.g. not during mornings) and day of week impact outcomes.

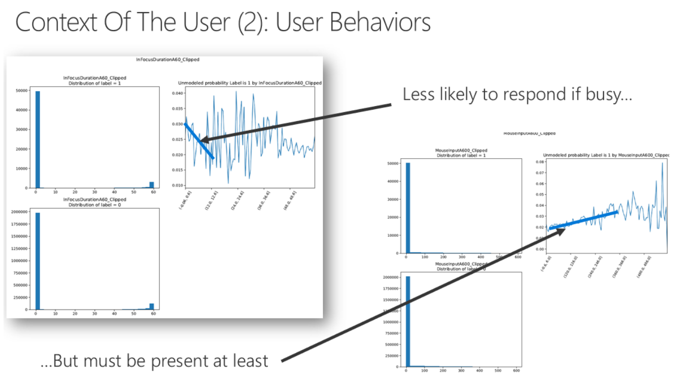


Fig. 8. Insights about user interactions for the SIUF ML model, showing the complex impact of user interactions.

first (bugs can happen), unforeseen situations and inputs, interacting with other software, and unpredictable human reactions. These reasons are refrains of reasons to conduct A/B testings for other software. Overall, our experiences are that ML-driven software are still software, and best practices like A/B testing are still beneficial. While these reasons could be raised even without our case studies, our empirical evidences show that these can (and do) occur in practice.

A. *Safety First: Bugs Can Happen*

ML models, by themselves, do not produce user-facing behaviors in our case studies (e.g. initiating reboots or displaying surveys); to do so, other codes have to call the ML model and then take its outputs to perform actions. Those codes can have bugs. Consequently, as with other types of software, A/B testing helps to ensure quality. For both cases of our ML-driven software features, A/B testing helped to uncover quality issues, which were then addressed.

B. *Unforeseen Situations and Inputs*

For real-world software (especially the Windows operating system), it is known that many unforeseen usage situations/contexts are possible and that a complete in-house testing is impractical [38]. Therefore, the use of real-world A/B tests helps to ensure quality. This is particularly true for ML-driven software, since the motivations/justifications for using ML are complex and not well covered by existing rules. Real-word A/B testing is potentially even more important since it can be hard to foresee the behaviors of ML-driven software in complex situations.

A particularity of ML-driven software is a mitigation plan for unforeseen bad inputs. Well-formed data are required for ML models, and significant efforts in data science are spent on sanitizing/cleaning data [39]. Ensuring all the different ways (possibly unforeseen) that bad inputs are correctly mitigated (e.g. clipping extreme values, creating a special category for unrecognized strings) is another reason for the need for A/B testing.

C. *Interacting with Other Software*

Evolving software to be ML-driven (perhaps more so than software built from the ground up to be ML-driven) entails working well with other software components as well as with other parts of the ML platform.

As we will discuss in more detail in the next section, an effective approach to evolve existing software to be ML-driven is to *add* the ML model to the existing rules. This implies that the ML functionality has integrated appropriately with other parts of the software. For example, in ML-driven WU, when the ML model defers a reboot, integration with a retry logic is needed to ensure that critical updates are installed in a timely manner by retrying reboots as soon as possible.

Additional components are also needed to ensure that the ML-driven software works as expected with the ML platform. For example, for both software features, additional validations were needed to verify telemetry reporting and model updates.

A/B testing, with its ability to isolate the impact of a change, is especially useful. For the Windows operating system, which is large and complex, attributing and debugging problems can be difficult [17]. Therefore, the use of A/B testing helps to isolate issues in the system caused by adding the ML functionality.

D. *Unpredictable Human Reactions*

ML-driven software aim to improve user experiences. However, positive user reactions are not assured. This is especially true in both of our cases, which tackle customer pain points that might have caused negative reactions. A/B testing is a proven approach to successfully build software with potentially unpredictable user reactions. In both of our cases, we used various metrics to measure user reactions in our A/B tests. They yielded interesting insights of user reactions to our changes, which we discuss in more detail in Section VIII-C.

VII. Insights of Effectively Building ML-driven Software Using A/B testing

Our experiences of evolving software to be ML-driven yielded several insights about how to effectively build ML-driven software, particularly using A/B testing.

A. *Make the ML Model an Addition*

Since evolving software to be ML-driven usually involves starting with the existing functionality that has rules and conditions, we found that adding the ML model on top of existing system is an effective incremental approach. It reduces both engineering risks as well as organizational risks.

For our software features, replacing all the rules and conditions by ML models would have been much larger scoped and more error-prone, which entails risks not only from adding codes but also from removing codes. For example, for WU, removing the ”Active Hours” functionality would have entailed changing UI elements, shown in Figure 1, as well as behaviors. By adding the ML model as another check, the risks were lowered. This enabled engineering team to successfully build and deploy with high quality, and then start the work to retire/migrate other rules and conditions.

Another factor is the organizational risk. With limited resources and many innovation investment options, committing significant resources can be risky. Initial successes and evidence of efficacy help to gain organizational support for more significant investments. For example, in the ML-driven WU, in addition to the positive customer reviews, metrics from telemetries also demonstrated benefits. Data from when reboots eventually take place indicate that the default 8-5 active hours may not be optimal and that ML-driven functionality can help identify better reboot times. These information helped to gain organizational support for further investments.

B. *Use Overall Success Metrics in Addition to Scenario Success Metrics*

In addition to scenario success metrics, overall success metrics are needed in A/B testing because ML can cause

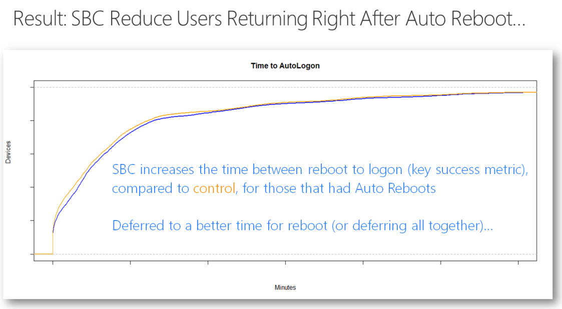


Fig. 9. Evaluation of Time to Logon for ML-Driven WU, where an SRM occurred due to the ML model.

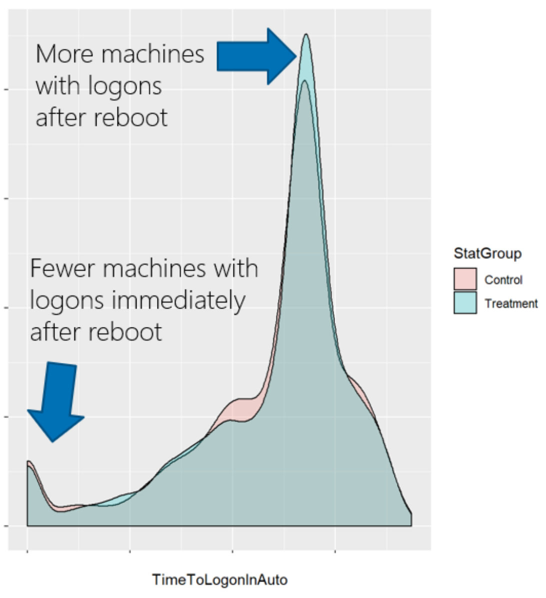


Fig. 10. Improvement in Time to Logon After Automatic Reboot due to the ML update for WU.

sample ratio mismatches (SRMs). An SRM in A/B testing is where the sample size is statistically significantly different between different groups. For example, in a 50/50 A/B test of 100 devices, 0 and 50 devices reporting a metric would constitute an SRM for the metric. Traditionally, this is a problem because it is unknown what other devices would have reported data. Therefore, any comparison of the metric between A/B groups may be invalid due to the fact that the difference could be caused by the devices selected rather than the software feature change. However, the problem lies in that the ML-driven software feature or the use of ML can cause SRMs *on purpose*.

Both of our ML-driven software features aim to avoid potentially disruptive behaviors (a bad time to reboot or to survey users). However, this implies that the metric for the scenario (e.g. whether the reboot disrupted the user or whether the user responded to the survey) does not exist when the ML model intervenes, causing an metric-level SRM. For example, for the ML-driven WU, results showed statistically significant improvement in increasing the time until users return to using their devices after the automated reboot (i.e. removing disruptive reboots), shown in Figure 9; however, the number of devices having automated reboots (and thus reporting the ”Time to AutoLogon” metric) were also statistically significantly lower. This was expected, as discussed in Section V-A, that more reboots were initiated by explicit user actions. Nevertheless, this SRM left open the possibility that those devices would have changed results. Our experience with this situation is to use overall success metrics in conjunction with the scenario success metrics to holistically assess the software feature change and triangulate on the impact to user experiences. In this manner, the ML-driven software feature can be shown to benefit those that it changes behaviors for as well as the overall success of the scenario. For ML-driven WU, in addition to time to user interaction after automatic reboots, we also used the overall success metrics, including proportion of devices upgraded and overall user satisfactions with Windows.

C. Make ML Models Updatable, Separate From Code Updates

While there are some practitioners who believe in evergreen ML models, e.g. ML burned onto silicon [1], our experiences are that updatable ML models separate from other engineering codes are preferable. In our ML-driven software features, the ML models are system resource files that can be updated independently from code updates.

We found this approach to be preferable for three reasons. First, this expedites engineering efforts by enabling concurrent training/retraining of the ML model and the coding/debugging of the software feature that utilizes the ML model. Second, we found that ML models usually require updating. For example, the WU ML model needed to be updated because it was initially built using data from Beta users. Figure 10 shows the statistically significant improvements of reducing user interruptions in the controlled rollout evaluation of the WU ML model update. Finally, related to the previous, updating engineering codes is more risky than updating a system resource file. By compartmentalizing the ML model, risks and costs associated with updating the ML-driven software feature are significantly reduced.

D. Avoid Complex Black-box Models

Recently, deep learning models have been picking up momentum and shown to be effective in numerous applications, such as facial recognition [6], text analysis [3], and voice recognition [7]. Our experience with ML-driven Windows features, which involve tabular telemetry data as inputs (such as day of week or seconds of interactive usage) rather than images, texts, or audio, indicate that unless the scenario is known to benefit from deep learning models (like the ones mentioned), traditional ML models are preferred. There are three reasons.

First, though there have been advancements in ”explainable” deep learning models [40], tools and practices for understanding traditional ML models are significantly more advanced [41]. In analyzing A/B tests of ML-driven software, we've needed to analyze the ML models in order to understand why they produced certain predictions and which inputs/factors lead to those predictions. In those situations, a traditional ML model like tree ensembles [42] provides much rich and mature explainability. These benefits are needed to debug potential data issues, help ensure fair and equitable ML models, as well as to provide transparency to engineering teams.

Second, deep learning models are larger on the disk. There have been recent advancements to reduce the size of deep learning models, which include utilizing a more compact architecture [43] and techniques like pruning and quantization [44]. However, the complexity of deep learning models particularly on tabular data still makes them less efficient in model size comparing to traditional ML models. The Windows operating system closely monitors the sizes of system files (which include ML models) and aims to minimize them on users' devices. As the number of ML-driven software features (and the number of ML models) grows, the need for parsimonious models also grows. Furthermore, since updating ML models is an integral part of the evolving process, model transmission costs are also of importance. Though the ML model files are relatively small in size, when scaled out to potentially billions of PCs (and updated regularly), the costs become non-trivial.

Finally, simpler ML models run faster, without requiring specialized hardware (e.g. GPUs). This becomes increasingly important as we look forward to more aspects of user experiences to be ML driven. Our Windows customers may choose different types of devices for various purposes (e.g. gaming, education). The need for efficient models that can enable effective ML-driven functionalities even in low-cost and/or low-powered devices becomes critical to ensure optimized experiences for all Windows users.

VIII. Real-world Challenges for A/B testing and ML-driven Software

In this section, we discuss three on-going challenges we face in evolving software to be ML-driven using A/B testing. We hope to tackle these challenges in the future, possibly through collaborations with researchers and other practitioners.

A. *Different Definitions of Equality for Different Applications*

Machine learning can make software less intrusive, more informative, and more reliable; however, using ML to change behaviors of an application can have unintended consequences of unfairly impacting its users [45]–[47]. Building fair machine learning systems is a difficult socio-technical effort that requires us to carefully define "fairness" in each scenario. Defining fairness is difficult in general but can be especially difficult when trying to ensure a complicated ML system is "fair" to all users.

For our ML-driven WU model, our model rollout data suggested that the model impacted laptops differently from desktops. As our goal is to be less intrusive for all users rather than just desktops, we are interested in success metrics that are approximately equal for both laptops and desktops. This notion of a fair model is common in the algorithmic fairness literature [48]. Many of these fairness definitions balance some metric for success across different subgroups. Notions like accuracy parity or false positive parity are similar to our desire for similar average values of the metric TimeToLogon between laptops and desktops. We addressed the disparity in TimeToLogon by training two separate models for the two device types. This example highlights one of the challenges for building fair machine learning systems. We may not know what subgroups will be impacted unfairly before training and deploying the model. The literature often consider user subgroups that are important to the society like age, socio-economic status, gender, ancestry and so on. However, the software and the ML model might unfairly impact unanticipated subgroups like laptops vs. desktops or region A vs. region B. We can proactively mitigate some sources of unfairness, but this example highlights the need for A/B testing and retroactive investigations.

For our ML-driven SIUF model, we observed that the model made SIUF less intrusive but it did so by choosing to show the prompt less often to older devices. Our goal was to be less intrusive and to give every user the opportunity to offer feedback. Translating these priorities into a quantitative notion of fairness was challenging due to the number of ways we could formulate the problem. We used the feature DaysSinceOOBE, which measures number of days since the Windows' out of box experience, as a proxy for device age. As this feature value got larger the probability of showing SIUF decreased. In this scenario, we did not have well defined groups that would allow us to use fairness notions based on equal success metrics for each subgroup. We could create subgroups by bucketing devices or we could draw on new ideas like individual fairness [49]. Individual fairness says that similar users should have a similar probability of getting the prompt. Although we thought this is an appropriate definition of fairness for our scenario, it highlights one of the difficulties of implementing fair ML-driven software. We realized that we must consider not just the definition of fairness but also the corresponding mitigation strategies. Given our system, it is easier and more effective to bucket the devices into subgroups based on device age and then re-weight the training data in a manner that produces similar probabilities for subgroups [50]. Individual fairness does not offer a clear mitigation strategy even though it is a reasonable definition of "fairness" in this scenario. Fair ML requires careful thoughts about defining, measuring, and mitigating unfairness and in practice we find that each of these three steps can impact the other two.

Microsoft is committed to the advancement of AI driven by ethical principles that put people first [51]. Practical implementations have many challenges. We found that no single definition of fairness works for all scenarios and that identifying the correct one depends on the software involved, the ML model used, the evaluation tools available to us (like our A/B testing platform), and the possible mitigation strategies. While there are some tools available to help implement mitigation strategies and academic literature available to help define fairness and fairness metrics, we found that building fair ML systems is a difficult task that can change drastically from scenario to scenario. More sharing of experiences and best practices by fellow practitioners may help.

B. Federated Learning with Local Privacy

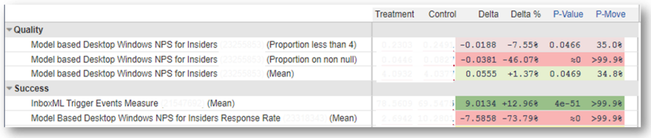
Federated learning (FL) with local privacy (LP) is a promising approach of distributing the costs of training ML models and strengthening the privacy protection of user data [3]. Using methods like Secure Multiparty Communications and Local Differential Privacy, researchers have shown that edge devices can collaboratively train an ML model without users' data leaving their devices. With increasing computing capabilities of edge devices and rising concerns over user data privacy, FL with LP has been receiving growing interests and has been investigated in many applications [23].

The challenge is how to incorporate FL with LP while retaining the benefits of A/B testing. FL with LP aims to minimize the data exposures of users. For example, in Secure Multiparty Communications [23], it is possible to compute a model update without revealing any information about the performance of the model on any individual device. However, anonymity hinders A/B testing and may increase difficulties in identifying issues for at-risk sub-populations. For example, all our efforts at testing and validating ML-driven software for bias and fairness require knowing the sub-population membership of those devices/users. Without the information, it may not be possible to ascertain whether issues exist (or have been addressed by fixes), since inherent population imbalances (e.g. more data from men than women [7]) may mask experiences of at-risk sub-populations. Approaches of securely sharing information about sub-population membership may be interesting areas of future research.

C. Pre-release A/B Testing May Not Be Indicative of Performance in Production

Pre-release A/B testing of ML-driven software have many benefits, as detailed in Section VI; however, making decisions based on data from pre-release A/B testing can be challenging due to differences relative to the general production. These differences include both the Beta users as well as the pre-release environment.

In the pre-release A/B test of ML-driven WU, there was a statistically significant drop in the proportion of devices automatically rebooted. This was linked to two phenomenons which were likely specific to the pre-release environment. First, it was not unusual for devices to be in the pre-release environment for only a short period of time (for testing and for trying out new software features, etc.); thus, some devices for which ML-driven WU deferred a reboot were not seen again (i.e. not getting another opportunity to complete the update). This is uncommon in the general population. In addition, the drop in automated reboots but more overall update completions meant that many users were initiating reboots themselves (after ML-driven WU deferred the automatic reboot initially). This was likely due to the particularities of Windows Insiders. Since the pre-release environment frequently gets updates with the latest new functionalities (e.g. weekly), which commonly require reboots, Windows Insiders (who also tend to be advanced users) seek and complete updates. This behavior is also uncommon in the general population. Nevertheless, the A/B test was able to identify these potentially negative reactions, which may have otherwise gone unnoticed, since the rate of automated reboots in the pre-release environment can fluctuate (e.g. the time of year effects and as the release gets closer to finalization).



		Treatment	Control	Delta	Delta %	P-Value	P-Move
Quality							
Model based Desktop Windows NPS for Insiders (23255853)	(Proportion less than 4)	0.2303	0.2491	-0.0188	-7.55%	0.0466	35.0%
Model based Desktop Windows NPS for Insiders (23255853)	(Proportion on non null)	0.0446	0.0827	-0.0381	-46.07%	≈0	>99.9%
Model based Desktop Windows NPS for Insiders (23255853)	(Mean)	4.0932	4.0377	0.0555	+1.37%	0.0469	34.8%
Success							
InboxML Trigger Events Measure (21547692)	(Mean)	78.5609	69.5475	9.0134	+12.96%	4e-51	>99.9%
Model Based Desktop Windows NPS for Insiders Response Rate (23318343)	(Mean)	2.6942	10.2801	-7.5858	-73.79%	≈0	>99.9%

Fig. 11. User behavior with ML-driven SIUF in pre-release A/B tests

ML-driven SIUF had similar challenges. While quality was high, the A/B tests had contextual complications that lead to equivocal results about user behaviors. For example, while the user response rate and satisfaction score increased, the total number of responses dropped significantly. A contribution factor was the resetting of the minimum usage condition upon new releases, which is much more frequent in the pre-release environment (as well as short-lived devices discussed previously). Therefore, deferring the SIUF significantly reduced the number of users surveyed, as the relevant metrics on A/B testing scorecard shown in Figure 11.

Discerning which issues are real and which are by-products of the pre-release context (thus safe to proceed) can be challenging. We found no simple rules-of-thumb, and each scenario involves nuanced interplay of users, context, and software feature functionalities, which requires significant time and efforts to understand. More experience reporting and knowledge sharing may help yield a generalizable guidance.

IX. Conclusion

Incorporating intelligent behaviors into existing software, driven by machine learning, can improve user experiences. The knowledge reported in this paper may help practitioners and researchers innovate (or start investing in) evolving software to be ML-driven, with the potential to greatly benefit our increasingly software-dependent society.

Acknowledgment

We would like to thank our colleagues and partner teams at Microsoft who made this work possible.

References

[1] A. Karpathy, "Software 2.0," 2017. [Online]. Available: https://medium.com/@karpathy/software-2-0-a64152b37c35

[2] T. Warren, "Windows 10 now uses machine learning to stop updates installing when a PC is in use," 2018. [Online]. Available: https://www.theverge.com/2018/7/25/17614842/microsoft-windows-10-updates-reboot-pc-machine-learning-feature

[3] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage, "Federated Learning for Mobile Keyboard Prediction," *arXiv preprint arXiv:1811.03604*, 2018.

[4] H. Li, K. Ota, and M. Dong, "Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing," *IEEE Network*, vol. 32, no. 1, pp. 96–101, 2018.

[5] S. Samarakoon, M. Bennis, W. Saad, and M. Debbah, "Federated Learning for Ultra-Reliable Low-Latency V2V Communications," in *Proc GLOBECOM '18*, 2018, pp. 1–7.

[6] J. Buolamwini and T. Gebru, "Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification," in *Conference on Fairness, Accountability and Transparency*, 2018, pp. 77–91.

[7] R. Tatman, "Gender and Dialect Bias in YouTube's Automatic Captions," in *Proceedings of the First ACL Workshop on Ethics in Natural Language Processing*, 2017, pp. 53–59.

[8] A. K. Elluswamy, M. Bauch, C. Payne, A. Karpathy, D. Shroff, A. Ramanandan, and J. R. H. Hakewill, "Predicting Three-dimensional Features for Autonomous Driving," 2020.

[9] G. Panger, "Reassessing the Facebook Experiment : Critical Thinking About the Validity of Big Data Research," *Information, Communication & Society*, vol. 19, no. 8, pp. 1108–1126, 2016.

[10] R. Kohavi, B. Frasca, T. Crook, R. Henne, and R. Longbotham, "Online Experimentation at Microsoft," *Data Mining Case Studies*, vol. 11, no. 2009, 2009.

[11] R. Kohavi and S. Thomke, "The Surprising Power of Online Experiments," *Harvard Business Review*, vol. 95, no. 5, p. 74, 2017.

[12] B. Ding, H. Nori, P. Li, and J. Allen, "Comparing Population Means under Local Differential Privacy with Significance and Power," in *Proc AAAI '18*, 2018.

[13] P. Dmitriev, S. Gupta, K. Dong Woo, and G. Vaz, "A Dirty Dozen: Twelve Common Metric Interpretation Pitfalls in Online Controlled Experiments," *Proc KDD '17*, pp. 1427–1436, 2017.

[14] H. Hohnhold, D. O'Brien, and D. Tang, "Focusing on the Long-term: It's Good for Users and Business," in *Proc KDD '15*, 2015, pp. 1849–1858.

[15] Y. Xu, N. Chen, A. Fernandez, O. Sinno, and A. Bhasin, "From Infrastructure to Culture: A/B Testing Challenges in Large Scale Social Networks," in *Proc KDD'15*, 2015, pp. 2227–2236.

[16] S. Gupta, L. Ulanova, S. Bhardwaj, P. Dmitriev, P. Raff, and A. Fabijan, "The Anatomy of a Large-Scale Online Experimentation Platform," in *Proc ICSA '18*, 2018.

[17] P. L. Li, P. Dmitriev, H. M. Hu, X. Chai, Z. Dimov, B. Paddock, Y. Li, A. Kirshenbaum, I. Niculescu, and T. Thoresen, "Experimentation in the Operating System: The Windows Experimentation Platform," in *Proc ICSE '19*, 2019.

[18] G. Saint-Jacques, A. Sepehri, N. Li, and I. Perisic, "Fairness Through Experimentation: Inequality in A/B Testing as an Approach to Responsible Design," *arXiv preprint arXiv:2002.05819*, 2020. [Online]. Available: http://arxiv.org/abs/2002.05819

[19] R. Kohavi, A. Deng, R. Longbotham, and Y. Xu, "Seven Rules of Thumb for Web Site Experimenters," in *Proc KDD '14*. New York: ACM Press, 2014, pp. 1857–1866.

[20] J. Vanschoren and H. Blockeel, "A Community-Based Platform for Machine Learning Experimentation," in *Proc KDD '09*, 2009.

[21] E. Alpaydin, *Introduction to Machine Learning*. MIT Press, 2020.

[22] S. Silva, B. A. Gutman, E. Romero, P. M. Thompson, A. Altmann, and M. Lorenzi, "Federated Learning in Distributed Medical Databases: Meta-Analysis of Large-Scale Subcortical Brain Data," in *Proc SBI '19*, 2019, pp. 270–274.

[23] L. Huang, A. L. Shea, H. Qian, A. Masurkar, H. Deng, and D. Liu, "Patient Clustering Improves Efficiency of Federated Machine Learning to Predict Mortality and Hospital Stay Time using Distributed Electronic Medical Records," *Journal of Biomedical Informatics*, vol. 99, p. 103291, 2019.

[24] S. O. Arik and T. Pfister, "TabNet: Attentive Interpretable Tabular Learning," *arXiv preprint arXiv:1908.07442*, 2019.

[25] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated Learning: Challenges, Methods, and Future Directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[26] N. Kilbertus, M. Rojas-Carulla, G. Parascandolo, M. Hardt, D. Janzing, and B. Schölkopf, "Avoiding Discrimination through Causal Reasoning," in *Proc NIPS '17*, 2017, pp. 656–666.

[27] S. Corbett-Davies and S. Goel, "The Measure and Mismeasure of Fairness: A Critical Review of Fair Machine Learning," *arXiv preprint arXiv:1808.00023*, 2018. [Online]. Available: http://arxiv.org/abs/1808.00023

[28] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. Tygar, "Adversarial Machine Learning," in *Proc AISec '11*, 2011, pp. 43–58.

[29] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecny, S. Mazzocchi, H. B. McMahan, T. Van Overveldt, D. Petrou, D. Ramage, and J. Roselander, "Towards Federated Learning at Scale: System Design," *arXiv preprint arXiv:1902.01046*, 2019.

[30] M. Zinkevich, "Rules of Machine Learning : Best Practices for ML Engineering," 2018.

[31] P. L. Li, J. Liao, B. Paddock, X. Chai, M. Kang, F. Campbell, I. Niculescu, N. Abburu, and J. Dooley, "Computing System for Training, Deploying, Executing, and Updating Machine Learning Models," 2020.

[32] D. Halfin and B. Lich, "Build Deployment Rings for Windows 10 Updates," 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows/deployment/update/waas-deployment-rings-windows-10-updates

[33] E. Bott, "Windows 10 Telemetry Secrets: Where, When, and Why Microsoft Collects Your Data," 2016. [Online]. Available: https://www.zdnet.com/article/windows-10-telemetry-secrets/

[34] T. Myerson, "An Update on What's Coming Next for Windows Insiders," 2017. [Online]. Available: https://insider.windows.com/en-us/articles/update-whats-coming-next-windows-insiders/

[35] P. Runeson and M. Höst, "Guidelines for Conducting and Reporting Case Study Research in Software Engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[36] R. T. Wickham, V. Deo, S. U. Khan, S. Sardesai, and A. D. Welker, "Software Updating System and Method," 2003.

[37] S. Sawaya, "How Windows Insider Feedback Influences Windows 10 Development," 2015. [Online]. Available: https://blogs.windows.com/windowsexperience/2015/06/12/how-windows-insider-feedback-influences-windows-10-development/

[38] P. L. Li, M. Ni, S. Xue, J. P. Mullally, M. Garzia, and M. Khambatti, "Reliability assessment of mass-market software: insights from Windows Vista®," *Proc ISSRE '08*, pp. 265–270, nov 2008.

[39] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 12–23.

[40] S. O. Arik and T. Pfister, "Tabnet: Attentive interpretable tabular learning," 2020.

[41] S. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

[42] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Advances in Neural Information Processing Systems 30*, 2017.

[43] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

[44] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2016.

[45] K. Holstein, J. Wortman Vaughan, H. Daumé III, M. Dudik, and H. Wallach, "Improving fairness in machine learning systems: What do industry practitioners need?" in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–16.

[46] J. Zou and L. Schiebinger, "Ai can be sexist and racist—it's time to make it fair," 2018.

[47] M. Kearns, "Fair algorithms for machine learning," in *Proceedings of the 2017 ACM Conference on Economics and Computation*, 2017, pp. 1–1.

[48] A. Chouldechova and A. Roth, "The frontiers of fairness in machine learning.(oct," *arXiv preprint cs.LG/1810.08810*, 2018.

[49] C. Dwork and C. Ilvento, "Individual fairness under composition," *Proceedings of Fairness, Accountability, Transparency in Machine Learning*, 2018.

[50] R. K. Bellamy, K. Dey, M. Hind, S. C. Hoffman, S. Houde, K. Kannan, P. Lohia, J. Martino, S. Mehta, A. Mojsilovic *et al.*, "Ai fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias," *arXiv preprint arXiv:1810.01943*, 2018.

[51] Microsoft, "Responsible AI," 2020. [Online]. Available: https://www.microsoft.com/en-us/ai/responsible-ai