Bighead: A Framework-Agnostic, End-to-End Machine Learning Platform

Eli Brumbaugh, Mani Bhushan, Andrew Cheong, Michelle Gu-Qian Du, Jeff Feng, Nick Handel, Andrew Hoh, Jack Hone, Brad Hunter, Atul Kale, Alfredo Luque, Bahador Nooraei, John Park, Krishna Puttaswamy, Kyle Schiller, Evgeny Shapiro, Conglei Shi, Aaron Siegel, Nikhil Simha, Marie Sbrocca, Shi-Jing Yao, Patrick Yoon, Varant Zanoyan, Xiao-Han T. Zeng, Qiang Zhu

Airbnb Inc.

***Abstract*—With the increasing need to build systems and products powered by machine learning inside organizations, it is critical to have a platform that provides machine learning practitioners with a unified environment to easily prototype, deploy, and maintain their models at scale. However, due to the diversity of machine learning libraries, the inconsistency between environments, and various scalability requirement, there is no existing work to date that addresses all of these challenges.**

Here, we introduce Bighead, a framework-agnostic, end-to-end platform for machine learning. It offers a seamless user experience requiring only minimal efforts that span feature set management, prototyping, training, batch (offline) inference, real-time (online) inference, evaluation, and model lifecycle management. In contrast to existing platforms, it is designed to be highly versatile and extensible, and supports all major machine learning frameworks, rather than focusing on one particular framework. It ensures consistency across different environments and stages of the model lifecycle, as well as across data sources and transformations. It scales horizontally and elastically in response to the workload such as dataset size and throughput. Its components include a feature management framework, a model development toolkit, a lifecycle management service with UI, an offline training and inference engine, an online inference service, an interactive prototyping environment, and a Docker image customization tool. It is the first platform to offer a feature management component that is a general-purpose aggregation framework with lambda architecture and temporal joins.

Bighead is deployed and widely adopted at Airbnb, and has enabled the data science and engineering teams to develop and deploy machine learning models in a timely and reliable manner. Bighead has shortened the time to deploy a new model from months to days, ensured the stability of the models in production, facilitated adoption of cutting-edge models, and enabled advanced machine learning based product features of the Airbnb platform. We present two use cases of productionizing models of computer vision and natural language processing.

***Index Terms*—end-to-end machine learning infrastructure, deep learning, automated feature engineering, lambda architecture, cloud native**

I. Introduction

The ubiquity of machine learning in the products and businesses in the technology industry today presents a growing variety of challenges for its practitioners. Each moment, a vast number of diverse machine learning models are being developed and deployed. Despite the prevalence of machine learning, it is still a lengthy and challenging task to develop a machine learning model from the ground up and deploy it to production. There are many stages in the lifecycle of a model, including data exploration, feature engineering, model training, tuning, deployment, serving, evaluation, and monitoring. Each stage is a distinct process on its own which poses unique problems. As such, work in this field requires a wide spectrum of skills and domain knowledge ranging from mathematical reasoning to rigorous engineering principles. A platform that provides automation, enforces best practises, and enables users to build, iterate on, productionize, and maintain healthy machine learning models is thus critical to the success of many organizations.

Despite rapid developments in the field, there still lacks a *framework-agnostic, end-to-end* machine learning platform, and existing solutions do not satisfy the needs of machine learning practitioners. First of all, many platforms lack advanced feature engineering capability, leaving many challenges unsolved in a stage in model development where many practitioners spend the majority of their time [1]. For example, it is crucial to have correct values for the features that correspond to the timestamp of the labels. This process, called temporal joins, prevents the situation of data leakage [2], that is, features incorrectly containing information on the labels because the former were observed *after* the latter. Another challenge is that, for features that are generated and consumed in real time, we need a framework that can process, aggregate, and join both offline and online data sources. This is not a trivial problem since aggregations and temporal joins need to be properly modeled in a principled way.

Moreover, existing platforms typically focus on supporting only one model framework, often leading to *tight coupling* between the modeling layer and the infrastructure layer. This limits the options for practitioners when they build models, and can prevent cutting-edge algorithms and techniques from being explored and adopted. It also creates a lock-in with certain frameworks and makes migrations difficult when these frameworks evolve or get deprecated.

Apart from the drawbacks of existing systems, we have identified the following four major overarching challenges when building a well designed machine learning platform.

First, it is common for model developers to spend a non-trivial amount of effort to iterate on models and take them to production. Cleaning up the code, writing applications to serve the model, and thoroughly testing changes are frequent tasks. In some cases, developers even have to re-implement

978-1-7281-4493-1/19/$31.00 ©2019 IEEE
DOI 10.1109/DSAA.2019.00070

the prototype models in another programming language or with another framework. This daunting list of tasks can add significant friction to the entire feedback loop, and if not done properly, can bring instability and cause incidents to the system. To allow fast iteration and boost productivity, it is vital for a platform to offer a **seamless** user experience from prototyping to production with as much automation as possible.

Second, the domain of machine learning is highly heterogeneous and ever-changing. Models using certain algorithms are typically built on structured data, and state-of-the-art deep learning models can leverage unstructured data such as texts, images, and videos, each of which require unique processing. Meanwhile, algorithms, frameworks, and platforms are constantly being released and updated. New compute resources such as GPUs and TPUs are increasingly required. For such a platform to be useful, it needs to be **versatile** by supporting major frameworks and various compute resources, being flexible to accommodate frequent changes, and being extensible to allow future additions. To achieve these goals, the platform needs to decouple infrastructure from the model frameworks and provide proper abstractions.

Third, models are moved across a diverse set of environments throughout their lifecycle. These environments can differ in numerous aspects, such as hardware, operating systems, versions of software dependencies, and sources of data. For example, the production environment is often vastly different from the prototyping environment. Data used for offline training often comes from a different source from online inference. Consequently, data produced by the model in production can be inconsistent with that produced during prototyping, leading to undesired situations such as incorrect results. It is therefore important to guarantee that the models are developed and productionized in a **consistent** setting and produce consistent results.

Fourth, the scales of the datasets, throughput, latency requirements, etc. all vary drastically from model to model, and can fluctuate greatly over time. A modern convolution neural network can easily consume thousands of times more resources than a simple regression model. A fraud detection model may require sub-second latency, whereas a sales forecasting model may only need to run once per month. While having as much computing power as possible is one way to solve the scaling problem, cost adds constraints on how many resources can be deployed at a time. The ability to **scale** horizontally and elastically in response to the change of the workload is thus critical to the stability, reliability, and cost effectiveness of the platform.

Several machine learning platforms have been built, such as TFX [3], MLflow [4], H2O.ai [5], and Michelangelo [6]; however, none of them can address all the challenges mentioned above. Therefore, we have built Bighead at Airbnb, an end-to-end machine learning framework that powers our data-driven business and products. This paper explores Bighead's overall design goals and architecture, details the problems that each individual component aims to solve, and finally outlines a vision for the future of machine learning infrastructure. Bighead is widely used at Airbnb, with a variety of models in production, and has been powering many product features based on cutting-edge machine learning algorithms and techniques.

II. Related Work

Many machine learning platforms have been developed at various companies. We briefly overview some major works in this section. TFX [3] is an end-to-end machine learning platform developed by Google, which spans from prototyping to production. It exclusively supports TensorFlow [7] as the model framework. Kubeflow [8] is also developed at Google, focusing on serving models in Kubernetes. MLflow [4] is developed and open sourced by Databricks. It is integrated with several cloud service providers, such as AWS and Azure. H2O [5] is a open source machine learning platform implemented in JVM with API libraries in several languages. Skymind Intelligence Layer [9], built on top of DeepLearning4J, offers model serving and scalability in its enterprise edition. Several in-house platforms cover many aspects of the machine learning workflow, such as Uber's Michelangelo [6], Facebook's FBLearner Flow [10], and Groupon's Flux [11]. However, these platforms are internal and not yet open sourced. Data Robot [12] is a popular proprietary system that offers features for automated machine learning. Several systems like Polyaxon [13], Comet [14], and Atalaya [15] provides model serving. Cloud service providers offer systems that enable the building, serving, and management of models, including Amazon's SageMaker [16], Microsoft Azure Machine Learning [17], and Cloudera Data Science Workbench [18].

We found that these platforms do not meet the need by the machine learning community for a framework-agnostic, end-to-end platform, for several reasons. First, many of them do not cover the end-to-end workflow. In particular, an important feature that most platforms lack is the integration with feature engineering and management, which is considered by some to be the most crucial part of machine learning [1]. As mentioned in Section I, there exist many challenging problems pertaining to this stage that a platform needs to solve.

Second, existing platforms focus on the support of one machine learning framework, thus not giving first-class support for or even precluding the use of others. Moreover, most of the frameworks are not designed in a flexible way, and substantial work would be required for customized features, such as integration with a particular organization's data warehouse, or enforcement of data privacy policies.

Lastly, some platforms are proprietary, and while they might have a more complete coverage for the workflow or popular frameworks, they cannot be leveraged by other organizations.

For the above reasons, we decided to build Bighead on our own, while leveraging existing open source technologies as much as possible, such as Apache Spark [19], Apache Flink [20], Apache Airflow [21], and Kubernetes [22]. Rather than stitching separately developed components together, Bighead

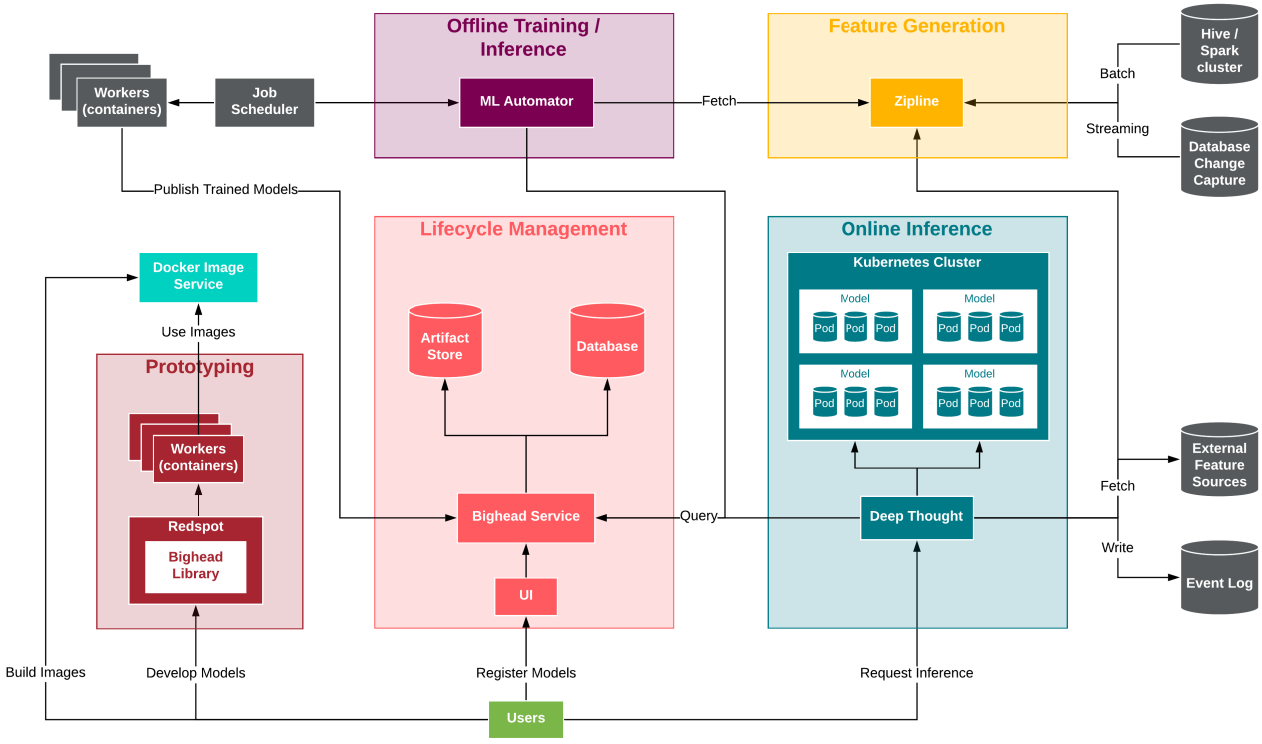

Fig. 1. An overview of the architecture of Bighead, depicting all the components.

is designed with the end-to-end workflow in mind from the start.

III. Design Goals and Architecture

A. Design Goals

To address the challenges in the domain of machine learning outlined in the previous sections, Bighead is designed to be:

Seamless: Offer a streamlined user experience from prototyping to production, across different frameworks. Enable users to easily write, iterate on, and deploy readable, robust, and reproducible machine learning models. Provide integration with existing data infrastructure and make data access and storage straightforward.

Versatile: Support all major machine learning frameworks, and make it easy to add support for new ones. Adhere to varying requirements, e.g. online and offline workloads, scheduled and ad hoc tasks, large data sizes, tight service level agreements, GPU computing, etc.

Consistent: Write once and use the same data sources, transformations, and environments to produce the same results in prototyping and production, training and inference, offline and online.

Scalable: Scale horizontally and elastically in response to the workload to handle changing requirements while being cost-effective.

These four design goals are not in isolation, but are connected to and reinforce each other. We will cover how they are applied to each stage in the end-to-end workflow in more details in the next section.

B. Architecture

Built with the design goals in mind, Bighead consists of multiple components (Figure 1):

- **Zipline**: A feature management framework with lambda architecture (Section IV)
- **Bighead Library**: A model development toolkit with a unified API that smoothly integrates popular machine learning libraries (Section V)
- **Redspot**: An interactive, multi-tenant prototyping environment (Section VI)
- **Docker Image Service**: An environment customization tool (Section VII)
- **Bighead Service**: A model lifecycle management service (Section VIII)
- **Bighead UI**: A user interface that provides easy operations and visualizations for models (Section IX)
- **Deep Thought**: An online inference service with containerization and cloud native architecture (Section X)
- **ML Automator**: An offline training and inference engine (Section XI)

Figure 2 depicts how each component is tailored towards a particular stage in the entire lifecycle of machine learning models. Users typically start their workflow with dataset exploration and generation using Zipline. Next, they prototype and train models in Redspot using Bighead Library, and build custom environments using Docker Image Service. Once the model is ready for production, they then use Bighead Service to keep track of their models, versions, and experiments. Finally, they use Deep Thought for online inference and ML Automator for offline training and inference. The components are well integrated with each other, but can also be used individually.

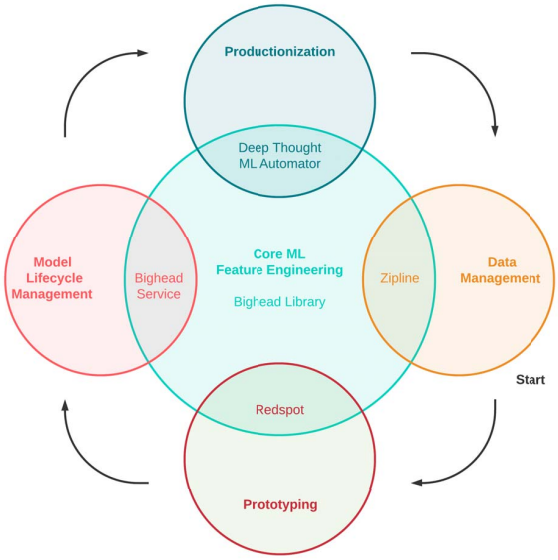

Fig. 2. Bighead's components mapped to each stage of an end-to-end workflow of machine learning model development.

In the following sections, we dive into each component of Bighead, in the order of the different stages of the end-to-end model lifecycle, and explain how the components address specific problems in each stage, and how they align with the four design goals.

IV. Zipline: Feature Management

The lifecycle of a machine learning model typically starts with collecting datasets comprised of *features* for the model to train and perform inference on, followed by constant iterations on processing the features to extract relevant information, called feature engineering. Developing and managing high quality of the feature sets is a hard task for model developers.

One of the most difficult challenges model developers encounter is to ensure *consistency* in the data across environments, particularly between offline training and online inference environments. A major concern for data consistency is *point-in-time correctness*, which is to correctly compute the value for a feature as of a precise point in time. This is important for models for which each row in the training dataset corresponds to an event that has occurred in the real world and thus has a timestamp (e.g. a user activity), as opposed to a static entity (e.g. an image). For example, for a model detecting logins from fraudulent users, the training dataset contains user login events. Each row has a label indicating whether the login is fraudulent behavior, along with the timestamp of the login. When constructing the training set, multiple features and the label are joined to form a complete vector. However, if not done properly, the formed vector can be incorrect when the values of the features are observed *after* those of the target variables. This can lead to a situation where the features carry information from the "future" of the label that the model would otherwise not have seen, typically dubbed "data leakage." For example, one of the features for the fraud detection model can be whether there is fraudulent behavior in the *previous* login session. Suppose the training set incorrectly uses a value of this feature that is observed *after* each login event corresponding to the label, then this feature effectively contains the label itself. As a result, the model can show overly optimistic predictive performance on the training data but performs poorly in production on inference data, where this feature does not contain this information any more.

Adding to the difficulty in ensuring consistency in the data is the complexity of computing features from aggregation, such as `last` and `sum`. Because aggregations can be expensive, they are usually performed by offline data processing frameworks. However, they are ill-suited for online inference. For features aggregated from real-time events, the values needs to be updated as new events arrive from online data sources. Due to the cadences of offline jobs, offline data sources can only produce snapshots at coarse-grained levels (e.g. daily or weekly), and thus are incapable of computing aggregated features in real time. For online inference requests *during the day*, two options are available for features computed offline: *end-of-day* values, or *start-of-day* values. On the one hand, *end-of-day* values contain future information and can cause data leakage. On the other hand, while using *start-of-day* values reduces the risk of data leakage, it deprives the model of recent data, which can often be very predictive. Aggregated features can thus be stale and bring inaccuracy to the results produced by the model.

Lastly, data scientists typically spend a significant amount of time developing a reliable feature set. To reduce work duplication, it is important to make the feature sets shareable and discoverable. A *feature store* of a shared repository of feature sets will suit this need in large organizations.

Zipline is a framework built to tackle these challenges in feature management for machine learning projects. It provides a general-purpose feature processing, aggregation, and joining framework that guarantees point-in-time correctness for the feature sets. It is built with the *lambda architecture* to combine offline batch sources and online streaming sources. Users only need to define features once, and Zipline makes them available in both inference and training environments, guaranteeing a single, **consistent** source of data for model consumption. It can efficiently backfill existing feature sets if new features are

added or more history is requested. Lastly, it offers a feature store for feature sharing, discovery, and reuse.

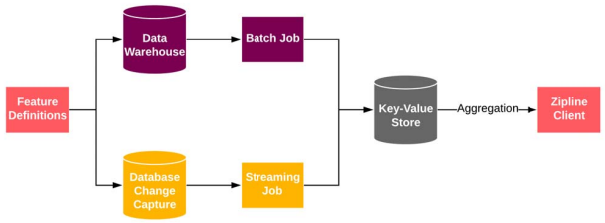

Fig. 3. Overview of the lambda architecture of Zipline.

A feature is the smallest piece of data available in Zipline, e.g. count of bookings made for a listing. A source is any store from which we can get data, for example, Hive or Kafka. There are two categories of sources: Event Source, for which the underlying data that generates the features is an immutable event with a logical timestamp for when it occurred in the real world, and Entity Source, for which the underlying data is a mutable entity that exists across time. A feature set is a group of features computed from the same source. A training set is a group of features from different feature sets joined together with a list of join keys, an ID, and a timestamp.

Zipline supports any aggregation operation that is commutative and associative (commutative monoids), including `last`, `first`, `min`, `max`, `count`, `sum`, and `average`. Optimizations are available if the aggregation is also invertible (Abelian groups). Windowed aggregation and bucketed aggregation are also supported.

Zipline's lambda architecture (Figure 3) consumes both daily offline batch uploads of feature values and raw events streamed from online sources. The batch uploads, typically running daily, produce partitions in the key-value store that contain the most up-to-date feature values for each day. The events are mutations to the feature values, including insert, delete, and update, and arrive continuously from streaming sources and are appended to the key-value store. With both offline and online sources, Zipline can produce feature values at many timestamps in the past by rolling up aggregations to these points in time in a correct and efficient manner. The aggregation is done by the Zipline Client class that reads events from the key-value store and computes the feature values upon request.

Zipline launches offline batch jobs that perform temporal joins, i.e. joining multiple feature sets to form the complete feature set, defined by a set of join keys and timestamps. For inference data, feature values are computed at the timestamp of inference requests. For training sets, the timestamps when the labels were observed are used. Temporal joins guarantees point-in-time correctness by ensuring that all feature values are the last values observed before the timestamp (Figure 4).

Zipline can handle complexities such as schema changes in the underlying data stores, allowing users to develop, test, and iterate on new feature definitions quickly. For offline sources,

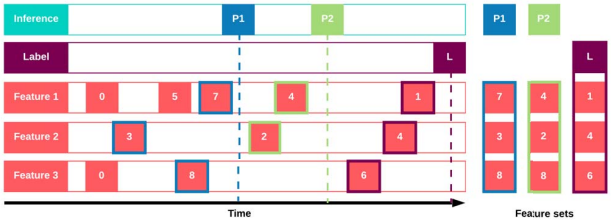

Fig. 4. An example of temporal join that guarantees point-in-time correctness. For inference request made at timestamps P1, the values of the features are the ones last observed before P1. The same happens for an inference request made at P2. For training data, the values of the features are the ones last observed before the timestamp of the label.

Zipline automatically detects and applies the schema change to add or remove features. For online sources, users can create the source with the new version of the schema and switch over once it is ready for consumption.

Finally, Zipline exposes an API for users to define a feature set. This API includes 1) metadata, such as owner and documentation, 2) source of the feature set, including a schema and relevant properties, and 3) a list of features and how to aggregate each of them. The feature set definitions are stored in a repository so that they can be easily discovered, inspected, and shared. Below is an example of a feature set definition. Zipline is aware of the timestamp of each raw event (ts), the primary key (listing), the aggregation (sum), and time windows (7d, 14d, etc.) for each feature.

```
owner: homes
source: {
  type: hive
  query: """
    SELECT
      n_bookings
    , n_nights_booked
    , listing
    , ts
    FROM data.bookings
    WHERE
    ds BETWEEN '{{start_date}}' AND '{{end_date}}'
"""
  dependencies: [data.bookings]
  start_date: 2019-01-01
}
operation: sum
windows: ["7d", "14d", "30d", "90d", "180d", "1y"]
features: {
  n_bookings: "Total bookings"
  n_nights_booked: "Total number of nights booked"
}
```

V. BIGHEAD LIBRARY: A MODEL DEVELOPMENT KIT

The development of a production-ready machine learning model is rarely as simple as selecting features and an algorithm. We observe that three challenges usually stand in the way of creating a production-grade model:

First, machine learning libraries usually have different APIs with different data input formats and requirements. These input requirements are usually tightly coupled with the framework

itself for performance reasons, and thus result in a large degree of boilerplate code whenever a new framework is used in order to explore a different method.

Moreover, because of the heterogeneity in machine learning framework APIs and input/output types, composability is rarely feasible, thus limiting the ability to share and reuse models and preprocessing pipelines. This leads to large amounts of duplicated work and difficulty in guaranteeing models can be saved and loaded correctly.

Finally, data is rarely clean and preprocesed in the ideal format. Images may need to be decompressed and resized, text may need filtering and tokenization, numeric data may need normalization and imputation, etc. Usually, libraries like Scikit Learn are used, but they tend to rely on either pure Python implementations or single-threaded numpy.

To address these problems, Bighead Library provides users with a **composable**, **consistent**, **versatile** interface for the creation of a self-contained model with minimal "glue" code. It achieves this through three key components:

First, the `Transformer` interface provides a way to define stateful or stateless functions that transform a collection of named *feature* tensors to another collection of named feature tensors. Features are not limited to machine types and can include strings, Python objects, etc. These `transformers` can be linked together into a directed acyclic graph (DAG) called a `Pipeline` to produce the desired combination of operations on a set of input data. Convenient `fit` and `transform` methods provide a familiar interface for training and inference of the `Pipeline`. Moreover, it handles serialization of underlying transformers. Given consistent input and output formats, `Transformers` and `Pipelines` are easily composable.

Second, Bighead Library includes a collection of useful CPU-tuned C++ accelerated implementations of common preprocessing steps like JPEG decoding/encoding, image resizing, normalization, categorical encoding, and text processing. These building blocks can outperform competing implementations vendored into specific machine learning frameworks. For example, for ResNet50 preprocessing steps, Bighead Library provides more than 40x speed-ups on a 64-core machine compared to Python PIL and MKL-linked numpy implementations by leveraging OpenMP, Xtensor [23], and Intel IPP. This ensures any machine learning framework can be swapped out with consistent results in the processing steps.

Third, Bighead Library provides lightweight wrappers over major machine learning frameworks, including Scikit Learn [24], MXNet [25], PyTorch [26], Tensorflow [7], XGBoost [27], spaCy [28] and others that standardize input and output formats. This makes it possible and convenient to mix and match different frameworks. For needs not covered by the built-in transformers and framework wrappers, users can easily define their own transformations in Python or C++ without needing to commit the code upstream. To add support for new frameworks, users only need to add wrappers for those frameworks that implement the `Transformer` API.

The primary implementation of Bighead Library is in Python. Experimental implementations in Scala and Clojure are also under development.

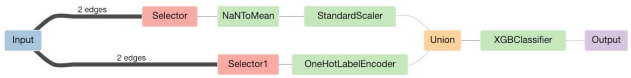

Fig. 5. A visualization of the Pipeline's DAG.

Below is an example of building a model that predicts whether a listing is a vacation rental, using four features: `num_bedrooms`, `num_bathrooms`, `room_type`, and `cancellation_policy`. `num_bedrooms` and `num_bathrooms` go through two steps, imputing with the mean value and standard scaling, and `room_type` and `cancellation_policy` are one-hot encoded. All features are then fed into an XGBoost classifier with 100 estimators, a learning rate of 0.1, and a max depth of 5. Figure 5 shows a visualization of the created pipeline.

```
# numeric features (numbers)
numeric = ['num_bedrooms', 'num_bathrooms']
# categorical features (low cardinality strings)
categorical = ['room_type', 'cancellation_policy']
# Create a new Pipeline instance
p = Pipeline(name='vacation_rental_listing')
# Impute NaN and perform normalization for numeric
p[numeric] >>= [NaNToMean(), StandardScaler()]
# One-hot encode categorical variables
p[categorical] >>= OneHotLabelEncoder()
# Feed features to the algorithm, set hyper-params
p >>= XGBClassifier(
    objective='binary:logistic',
    n_estimators=100,
    learning_rate=0.1,
    max_depth=5)

# Train the model
p.fit(training_set, labels)
# Perform inference
p.transform(inference_set)
```

VI. Redspot: Prototyping Environment

After data collection, the next stage in the model development workflow is prototyping. This step tends to be highly interactive, iterative, and exploratory, with many branching scenarios and short feedback loops. An ideal machine learning prototyping environment needs to provide 1) interactivity and feedback, 2) access to powerful hardware for compute-intensive models, and 3) easy access to data.

Redspot is a prototyping environment built on JupyterHub [29], with additional features including Docker integration, remote instance spawning, shared storage, and full integration with the data warehouse. It provides access to specialized hardware such as GPUs. Files are persisted across user sessions and shared between users via AWS EFS. It provides Jupyter Notebook UI which most model developers are familiar with.

Redspot is one major component that provides a **seamless** user experience. It is integrated with the rest of Bighead, including Bighead Service, Docker Image Service, and Deep Thought via APIs. Its UI is familiar to most users, which make model development easy. It promotes **consistency** by

ensuring that prototyping is done in the exact environment that the model will use in production, by using the same Docker image and running the same Pipeline code. It is **versatile** in that users can request custom, powerful hardware to suit their specific needs, for example, AWS P3 or X1 instances. They can also use customized dependencies in the Docker images, such as Python 3.6 with GPU-enabled Tensorflow 1.12.

VII. Docker Image Service: Environment Customization

Machine learning models have a diverse, heterogeneous set of dependencies. There needs to be an easy way to create their own runtime environments that are consistent with the rest of the infrastructure, without a limit on tools and frameworks. The environments should be composable so that they can be reused. Moreover, because the consistency and reproducibility of these environments are key to the reliability of the models running within, they need to be versioned in a way similar to software packages.

Docker Image Service is built on top of Docker to help solve these problems. As a configuration management solution, it provides 1) a composition layer on top of Docker, 2) a customization service that is user-facing, and 3) a repository of pre-built images. It promotes **consistency** and **versatility** throughout Bighead by creating a single Docker image that is carried through the system, from prototyping to production and across online and offline. Users can choose to use standard images, or compose custom images by combining layers that can have arbitrary dependencies. Semantic versioning is enforced on all images, so that development can be fast while the environment remains stable.

VIII. Bighead Service: Lifecycle Management

Once the model developers finish the prototyping stage, they are ready to deploy their models to production. Tracking machine learning model changes is as crucial as tracking code changes. Failure to properly version models that interact with production systems can lead to undesired situations. Also, comparing experiments and running burn-in tests before launching models into production is critical. A registry that clearly indicates the status of models ensures that changes are safe and reproducible.

Bighead Service is Bighead's lifecycle management service for machine learning models. It maintains a registry of the models in production, as well as the lifecycle status of each. It serves a single, **consistent** source of truth in the Bighead ecosystem on what models are deployed, and which version has been deployed and serving production traffic.

The object model of Bighead Service follows a hierarchy of `Model`, `Version`, and `Artifact`. `Model` is the top-level object that defines a machine learning project, including metadata such as name and owner, as well as configurations. Each `Model` can have multiple `Versions`, each corresponding to a snapshot of the code repository. Each `Version` can have multiple `Artifacts`, which are the serialized Pipeline instances produced by training. Training executes with the latest `Version` for each Model, and an `Artifact` marked as "active" will be used for inference. Via Bighead Library, Bighead Service stores additional metadata from the training process for human interpretation.

Bighead Service provides a set of API classes that the users can use at model development stage to specify operation configurations, such as compute resource, schedule, and data sources and storages for training and inference. These configurations are picked up during model registration and stored in the database, and are used in production without requiring any extra setups. It also exposes a set of REST APIs that allows users to programmatically manage model `Versions` and make `Artifact` deployments. The set of APIs ensure that users have a **seamless** process going from prototyping to production.

Below is an example of the class that the user writes that carries the configurations of a model, where training runs in a single machine using 16 cores and 32 GB of memory on Mondays, and inference runs in Spark using 16 executors everyday:

```
model = ModelConfig(
  name='vacation_rental_listing',
  owner='bighead',
  team='Airbnb',
  description=\
    'Predict if a listing is a vacation rental',
  tags=['xgboost'],
  pipeline=p, # define the Pipeline instance
  # define the data source and sink
  source=source,
  sink=sink,
  # define the schedule and resources for training
  fit_config=ScheduledComputeConfig(
    start_date='2019-01-01',
    single_machine_config=SingleMachineConfig(
      num_cpu=16, memory=32400),
      schedule=Schedules.Mondays),
  # define the schedule and resources for inference
  transform_config=TransformConfig(
    offline_config=ScheduledComputeConfig(
    start_date='2019-01-01',
    spark_config=SparkConfig(num_executors=16),
    schedule=Schedules.EveryDay),
  online_config=OnlineServingConfig(num_workers=4)))
```

IX. Bighead UI

To provide a smooth user experience, we built the Bighead UI as the main interface for our users to explore, interact with, and manage the lifecycle of their models.

Once models have been registered with Bighead Service, users can use the UI to track the models, deploy different versions of them, and monitor their status and performance. With the design goals in mind, we provide users with a **consistent** experience between the prototyping and production stages. More specifically, users can carry over highly customizable metadata from training to the Bighead UI, such as evaluation metrics and visualizations. For example, users can explore the model performance in the UI to determine which models should be deployed, via feature importance plots, ROC curves, and precision-recall curves. In addition, it serves as a convenient portal for model status with links to online and

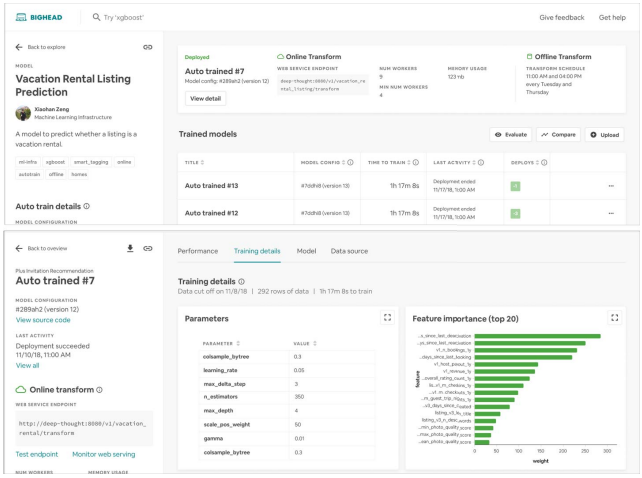

Fig. 6. Screenshots of UI showing an example Model, its Versions and Artifacts, and details of an Artifact.

offline serving engines. See Figure 6 for screenshots of the UI (note that the data is anonymized.)

X. Deep Thought: Online Inference

The last stage of the workflow is deployment to production. Online model serving is an essential aspect of model productionization. It allows live systems to consume model results and change behaviors accordingly. However, several difficulties exist that prevent online model serving from being consistent with the training stage.

First, differences can occur in the data source, model implementation, and environment. In offline training, data typically comes from data warehouses, whereas during online inference, data is usually supplied by the clients, such as other services or real-time events. Discrepancies such as incompatible types and missing values are possible, causing unexpected results from incorrect values to system errors. Model implementations can also be different, since it is possible for model developers to develop prototypes in languages such as Python and R, while production systems are built with Java or C++ so that the prototypes need to be rewritten. Finally, the environment can be different in terms of hardware, operating systems, library versions, etc. All these differences can potentially affect the behavior and performance of the models during online inference.

Second, taking model to production can require a significant amount of work. As a result, data scientists cannot launch models without support from engineering teams, and work can be delayed due to engineering resource limitation and prioritization. In less ideal cases, engineers need to rebuild models because of the aforementioned differences, causing further delays.

Finally, scalability is still a major concern. Resource requirements can vary across models, and there is rarely one setting that fits all. As a result, each model needs to be able to scale on its own. Resource isolation is also important,

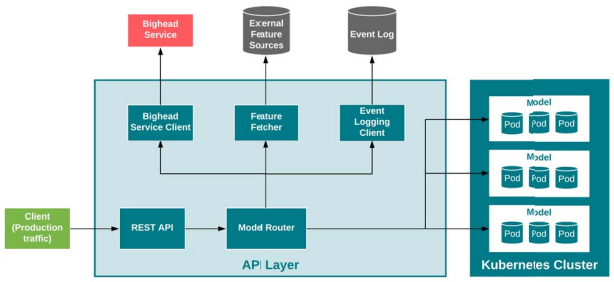

Fig. 7. An overview of the architecture of Deep Thought. It consists of a gateway API layer that routes requests to corresponding model pods, and a set of load-balanced, auto-scaled model pods in the Kubernetes cluster.

so that models will not interfere with each other. Also, because throughput can fluctuate greatly across time, to avoid unnecessary cost, the system needs to be elastic in response to the changes in the traffic.

Deep Thought is the online inference service component of Bighead. Built on top of Kubernetes, Deep Thought is a thin routing layer that provides REST endpoints around the transform function of Bighead's Pipeline API 7. It queries Bighead Service for the lifecycle status of Models and Artifacts, and maintains an in-memory pool of references to models that are deployed to a Kubernetes cluster and kept up-to-date with Bighead Service. Each model is deployed as a Kubernetes service consisting of a set of pods, and each pod is running a web service that provides REST API to the model Pipeline. By using Docker image and Bighead Library, Deep Thought guarantees the **consistency** of model results use of exactly the same data source, transformations, and environment as used in training.

Deep Thought supports two formats of payload, MessagePack and JSON. *MessagePack* is a binary serialization format. It is a fast and compact solution for use cases such as images, where encoding and decoding images in Base64 is slow, and embeddings, where serializing an array of floats into JSON string produces larger payloads. The MessagePack endpoint is optimized for production. The *JSON* endpoint is supported as it is a popular format, and is easy for testing and debugging.

Deep Thought can start serving traffic once the model has been deployed in Bighead Service, without extra configuration and deployment steps, offering a **seamless** user experience. It is well integrated into the event logging and metrics monitoring systems, as well as the other components of Bighead such as Zipline, to provide features that are critical for model serving in production:

Event logging. It provides an event logging interface that logs each transform request. The default implementation logs events to a log file, and it is easy to add other implementations such as logging to a Kafka topic.

Feature fetching. Deep Thought can read data from external sources via its Feature Fetcher interface. For example, a model can declare a list of Zipline Feature Fetchers. The incoming

request then only needs to provide the primary keys for each data source, and the corresponding feature fetcher will fetch the data and combine it with the payload before transforming it.

Metrics. Deep Thought provides a Metric Provider interface that provides Dropwizard metrics, with additional metadata such as model name attached to each metric. Available metrics include latency, QPS, and number of exceptions.

By leveraging Kubernetes, model pods can **scale** automatically in response to traffic. Resource segregation is guaranteed across models.

XI. ML Automator: Offline Training and Batch Inference

Apart from online model serving, automated offline training, inference, and evaluation are another prevalent use case for machine learning. Scheduling, orchestration, compute resource allocation, and result storage all need to be automated. In addition, retries, dashboards, and alerts are essential to the robustness of offline jobs.

ML Automator is the offline execution engine of Bighead built on top of Apache Airflow. It allows users to programmatically define and schedule offline jobs for models. It queries Bighead Service for the training, inference, and evaluation jobs it needs to schedule, and interacts with Airflow's scheduler to create DAG runs. Airflow promises retries on failed tasks and orchestrates the execution of inter-dependent tasks, and its UI serves as a dashboard for the users.

ML Automator promotes **consistency** by using Docker and Bighead Library to guarantee the use of the same data sources, transformations, and environments across the stack. It offers **seamless** model deployment by automating offline tasks for training, inference, and evaluation on given cadences and resources via Airflow DAGs. It is also integrated with Zipline for data sources. ML Automator achieves **scalability** by leveraging Spark for distributed computing for large datasets, or machines with powerful hardware such as CPU, memory, and GPUs. Users can specify the amount of resources in the model configuration file.

XII. Use Cases

Bighead has been widely adopted and powering numerous features across many teams at Airbnb. More than a hundred machine learning models across over a dozen teams have been built and productionized on Bighead. Bighead has also greatly enhanced user productivity. Models that used to take on average eight to twelve weeks to build now only takes a few days to develop and productionize. We highlight a couple use cases in the subsections below, particularly in computer vision and natural language processing.

A. Listing Image Room Type Classification

Listing photos are one of the most critical factors for decision-making during a guest's search journey at Airbnb. The Applied Machine Learning team productionized a model that categorizes the listing photos into different room types [30]. The model is a multi-class classifier built with Keras with TensorFlow as the backend. We used 2.5 million labeled images to train a ResNet50 model pre-trained with ImageNet using transfer learning. The model has been productionized with Bighead to classify two hundred million listing images on Airbnb, as well as newly uploaded images each day in both online and offline settings. It leverages Bighead Library's native image processing tools to gain a 40-time speed-up in the preprocessing step. The model makes possible a simple Home tour where photos with the same room type can be grouped together, as well as makes it much easier to validate the number of certain rooms and check whether the basic room information is correct. It has enabled features including image ranking and grouping for guests, enhanced host image uploading flow, photo review, and automated listing review.

B. Message Intent Understanding

Ensuring good communication between guests and hosts is one of the keys to a good guest experience. Millions of guests and hosts communicate on the Airbnb messaging platform about a variety of topics. To improve the experience for guests, three Airbnb teams – Shared Products, Applied Machine Learning, and Machine Learning Infrastructure – have developed a message intent classification model to automatically predict and understand the intent of guests' messages to hosts [31]. The model tries to understand the major topics of the intent behind each message. It is divided into two phases: Phase 1 is an LDA-based unsupervised model that uses NLTK, gensim [32], and spaCy [28]. Phase 2 is a CNN-based supervised model that primarily uses Keras. Separate models are built for each trip stage, including pre-booking, pre-trip, on-trip, and post-trip. Each trip stage has one million training samples for Phase 1 (four million messages in total), and 25–35k training samples for phase 2 (120k messages in total). The median of the message length is around 25 words, and the max length can reach 3,000 words where truncation is used. The models are developed in Redspot (prototyping component), and classifies around 40–50 messages per second using Deep Thought (online inference component). The model has greatly shortened the response time for guests and reduced the overall workload required for hosts. It has also enabled Airbnb to provide essential guidance and thus a seamless communication experience for both guests and hosts.

XIII. Conclusions and Future Work

Bighead is a framework-agnostic, end-to-end machine learning platform that is designed to be seamless, versatile, consistent, and scalable. It addresses many challenging problems in the model development and deployment workflow, and greatly enhances user productivity.

Bighead is an ongoing project, and many features have been planned and are being developed, including feature quality monitoring, distributed training, model performance monitoring, and automated deployment. A few experimental features are also in progress, including implementations of the model development library in C++ / Scala / Clojure,

an implementation of XNOR-Net [33], and advanced hyperparameter tuning using the Ray project [34]. In the future, we expect Bighead to offer more features that further increase the productivity of model developers, as well as improve support for major frameworks. We also plan to open source Bighead so that the greater community can benefit from and contribute to this work.

ACKNOWLEDGMENT

We thank Kedar Bellare, Cassie Cao, Pascal Carole, Jeff Chang, Robert Chang, Ashley Chen, Cindy Chen, Brendan Collins, Chris Goldammer, Mihajlo Grbovic, Thomas Legrand, Yuchen Liu, Chirag Mahapatra, Ben Mendler, Haggai Nuchi, Christopher Mitcheltree, Kartikeya Shandilya, Peggy Shao, Jiaying Shi, Teng Wang, Xingnan Xia, Bob Zheng, Chris Zhu for their partnership with, and support and feedback for Bighead. We thank Steve Flanders, Arjun Kumar, and Brian Wolfe for proofreading the paper.

REFERENCES

[1] P. Domingos, "A few useful things to know about machine learning," *Communications of the ACM*, vol. 55, no. 10, p. 78, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2347736.2347755

[2] S. Kaufman, S. Rosset, C. Perlich, and O. Stitelman, "Leakage in Data Mining: Formulation, Detection, and Avoidance Shachar," *ACM Transactions on Knowledge Discovery from Data*, vol. 6, no. 4, pp. 1–21, 2013.

[3] D. Baylor, E. Breck, H.-t. Cheng, N. Fiedel, C. Yu Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Yuen Koo, L. Lew, C. Mewald, A. Naresh Modi, N. Polyzotis, S. Ramesh, S. Roy, S. Euijong Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich Google Inc, "TFX: A TensorFlow-Based Production-Scale Machine Learning Platform," *KDD 2017*, pp. 1387–1395, 2017.

[4] M. Zaharia, "Introducing mlflow: an open source machine learning platform," https://databricks.com/blog/2018/06/05/introducing-mlflow-an-open-source-machine-learning-platform.html, 2018, accessed: 2019-01-18.

[5] "H2o.ai," https://www.h2o.ai/, 2019, accessed: 2019-01-18.

[6] J. Hermann and M. Del Balso, "Meet michelangelo: Uber's machine learning platform," https://eng.uber.com/michelangelo/, 2017, accessed: 2019-01-18.

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, R. Monga, S. Moore, D. Murray, J. Shlens, B. Steiner, I. Sutskever, P. Tucker, V. Vanhoucke, V. Vasudevan, O. Vinyals, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," in *USENIX Symposium on Operating Systems Design and Implementation*, 2015, p. 19. [Online]. Available: http://download.tensorflow.org/paper/whitepaper2015.pdf

[8] D. Aronchick and J. Lewi, "Introducing kubeflow - a composable, portable, scalable ml stack built for kubernetes," https://kubernetes.io/blog/2017/12/introducing-kubeflow-composable/, 2017, accessed: 2019-01-18.

[9] "Skymind intelligence layer: A platform for operating models in production," https://skymind.ai/platform, 2019, accessed: 2019-01-18.

[10] J. Dunn, "Introducing fblearner flow: Facebook's ai backbone," https://code.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/, 2016, accessed: 2019-01-18.

[11] D. C. Spell, X.-H. T. Zeng, J. Y. Chung, B. Nooraei, R. T. I. Shomer, L.-Y. Wang, J. C. Gibson, and D. Kirsche, "Flux : Groupon's automated , scalable , extensible machine learning platform," *Proceedings - 2017 IEEE International Conference on Big Data, Big Data 2017*, pp. 1554–1559, 2017.

[12] "Datarobot: Automated machine learning for predictive modeling," https://www.datarobot.com/, 2019, accessed: 2019-01-18.

[13] "Polyaxon: An open source platform for reproducible machine learning at scale," https://polyaxon.com/, 2019, accessed: 2019-01-18.

[14] "Comet," https://www.comet.ml/, 2019, accessed: 2019-01-18.

[15] "Atalaya: Deploy machine learning faster with ease," https://atalaya.io/, 2019, accessed: 2019-01-18.

[16] "Amazon sagemaker: Machine learning for every developer and data scientist," https://aws.amazon.com/sagemaker/, 2019, accessed: 2019-01-18.

[17] "Azure: Build, train, and deploy models from the cloud to the edge," https://azure.microsoft.com/en-us/services/machine-learning-service/, 2019, accessed: 2019-01-18.

[18] "Cloudera data science workbench: Self-service data science for the enterprise," https://blog.cloudera.com/cloudera-data-science-workbench-self-service-data-science-for-the-enterprise/, 2017, accessed: 2019-08-09.

[19] M. Zaharia, M. Chowdhury, T. Das, and A. Dave, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 2–2. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

[20] P. Carbone, S. Ewen, and S. Haridi, "Apache Flink: Stream and Batch Processing in a Single Engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pp. 28–38, 2015.

[21] "Apache airflow," https://airflow.apache.org/, 2019, accessed: 2019-01-18.

[22] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

[23] S. Corlay, J. Mabille, and W. Vollprecht, "C++ tensors with broadcasting and lazy computing," https://github.com/QuantStack/xtensor, 2019, accessed: 2019-01-30.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," pp. 1–6, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[26] A. Paszke, G. Chanan, Z. Lin, S. Gross, E. Yang, L. Antiga, and Z. Devito, "Automatic differentiation in PyTorch," *31st Conference on Neural Information Processing Systems*, no. Nips, pp. 1–4, 2017.

[27] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *KDD '16 Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 785–794.

[28] "spacy: Industrial-strength natural language processing in python," https://spacy.io/, 2019, accessed: 2019-01-18.

[29] "Jupyterhub: A multi-user version of the notebook designed for companies, classrooms and research labs," https://jupyter.org/hub, 2019, accessed: 2019-01-18.

[30] S. Yao, Q. Zhu, and P. Siclait, "Categorizing listing photos at airbnb," https://medium.com/airbnb-engineering/categorizing-listing-photos-at-airbnb-f9483f3ab7e3, 2018.

[31] M. Du and S. Yao, "Discovering and classifying in-app message intent at airbnb," https://medium.com/airbnb-engineering/discovering-and-classifying-in-app-message-intent-at-airbnb-6a55f5400a0c, 2019.

[32] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[33] A. F. Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," *Eccv*, pp. 1–17, 2016. [Online]. Available: https://arxiv.org/abs/1603.05279

[34] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, and M. I. Jordan, "Ray: A Distributed Framework for Emerging AI Applications," 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/nishihara