SCTE•ISBE CABLE-TEC EXPO® 2020
VIRTUAL EXPERIENCE » OCTOBER 12-15
2020 Fall Technical Forum
SCTE·ISBE • NCTA • CABLELABS®

Operationalizing Streaming Telemetry and Machine Learning Model Serving

Customer Experience Automation

An Operational Practice prepared for SCTE•ISBE by

Nick Pinckernell
Distinguished Engineer
Comcast
183 Inverness Dr W, Englewood CO 80112
nicholas_pinckernell@comcast.com

Scott Rome
Principle Researcher
Comcast
1800 Arch St, Philadelphia, PA 19103
scott_rome@comcast.com

Table of Contents

List of Figures

List of Tables

1. Introduction

Customer Experience, Telco, Machine Learning, Automation… all sounds a bit dry, no? If not, this should help shed light on a specific use case, but also provide details and lessoned learned while building the solution. If so, read on to perhaps find some surprising details. Many are aware of the impact from Machine Learning but may not be aware just how many portions of the enterprise it is now altering -- or the magnitude of those changes.

With building an ML platform in mind, specifically to improve and automate the customer experience, this paper will illustrate how we accomplished this. It will also detail lessons learned and insights not only on the data itself, but on the architecture and thinking behind it.

Another focus is the challenges and differences of scaling and maintaining the Machine Learning components of the architecture. The highlight here is that while Proactive Network Management (PNM) [1] has laid out a number of excellent methods on how to collect and analyze network telemetry from CPE and other headend equipment, it has not covered the aspects of what to do or how to handle the data with respect to utilizing ML models.

This paper will begin with the Customer Experience use case, and work from that point through to the end solution. A number of open source technologies are referenced as possible implementations for components. Other components can certainly be used.

The desire to share this information stems from the fact that some of these solutions are not easy. They require not only multiple resources to develop the front-end user interface, but the back-end platform as well. After the ML models have been trained, the task of building the platform, scaling, testing specific tools and gluing everything together is still rather manual and prone to performance and optimization challenges. With those things in mind, the details throughout this paper should aid the reader in determining directions to go when considering, building and scaling such a system.

2. History

Telemetry collection and polling have been widely covered in the industry with both positives and negatives. The positives almost always outweigh the negatives, in terms of providing the ability to determine outages quickly, gain insights on which partitions and elements in the network require maintenance and many other useful applications. Using telemetry with machine learning is also fairly old, just not commonly referred to as “machine learning”. Take adaptive equalization in digital communications as an example. The parallels are many between parts of adaptive equalization and how machine learning algorithms work. Or, take the DOCSIS upstream pre-equalization process as another example. For each burst of data in a transmission, the preamble is used just like a training set for a supervised ML algorithm, but for the CMTS’s upstream adaptive equalizer. Likewise for the downstream, with a blind-equalizer -- it is the same as an unsupervised ML algorithm. Overall, there are inputs, derived coefficients and an optimization problem – which is to minimize the mean square error of the outputs.

Applying ML algorithms to other telemetry angles is simply considering different shapes. With polling or collection systems, gathering telemetry from CPE, headend or data center equipment, and funneling that into distributed computing environments to do essentially the same thing the CMTS was doing with adaptive equalization, is a larger scale scenario with potentially more complex models. This method of analyzing telemetry is just the next evolution, after analyzing telemetry for outages or full spectrum data

for various distortions -- but now using more complicated methods such as ML models to tease out new artifacts and correlations in determining network health.

3. Customer Experience as it relates to ML

What is customer experience? According to customer experience futurist Blake Morgan, it "really boils down to the perception the customer has of your brand" [2]. In the use case laid out, it is the perception of the self-service customer experience.

3.1. Use Case

The outcome of the project is to automate simple common customer service questions and return immediately useful feedback or solutions to the posited issue or question. The goal here was two-fold: one, to improve the customer experience by knowing their issue before they do and if possible remedy it, and two, to reduce the number of calls to customer service.

When left open-ended, the outcome is intractable, given the large problem scope of all possible issues customers may face while going through all the various customer service avenues available to them. Therefore the scope and requirements of the outcome were reduced to answer a few basic questions, then continue to build out the capability to handle more queries from there. These initial questions were:

- Is there an issue with my internet service?
- Is there an issue with my video service?
- I have a billing or account inquiry?

All of the problems require use of machine learning. The first two require telemetry data and additional machine learning as well. For the third question, the goal was to handle simple questions initially, such as "why is my bill different than last month?" or "how much does my TV package cost?" and has since been expanded to handle more questions.

With the goals and outcome clearly defined, the solution required thinking on how to accomplish this, given the large number of telemetry and other data sources available.

4. Telemetry

The breadth of telemetry for most telecommunications companies and network operators is vast. Data comes from a number of sources which generally are customer devices, headend or data center equipment or public devices. Customer devices or CPE (Customer Premises Equipment) are usually comprised of cable modems, Wi-Fi Access points, cable boxes and the ever growing list of IoT devices. Central locations such as the headend, Point of Presence (POP) and data centers tend to be the geographic source for equipment such as CMTSs, access networks, video sources and more.

The correlation between our initial questions and data sources were fairly straightforward:

- Internet service problem: telemetry from the cable modems and CMTSs
- Video service problem: errors from our Video Backend Service (VBS)
- Billing / account inquiry: billing data system

During the exploration of datasets, a single dataset was found that provided attributes for both the internet and video service questions, which was errors from the OS that runs on both cable modems and STBs.

The informational and error messages from devices built on the Reference Design Kit (RDK) [3] turned out to be very valuable, as both the Set-top Box (STB) and Cable Modem (CM) use the same transmission medium from the premise to the headend or hub site.

4.1. Attributes

Telemetry from the RDK consists of a number of attributes useful to determine if there are issues from the CM side, STB side or both. It contains general information about the CPE:

- CPU utilization
- Memory utilization
- System load
- Wi-Fi signal strength
- Etc.

Perhaps the most useful data from RDK, however, are error codes and counts. These errors relate to dropped packets, firmware errors, signal errors, etc. The specific attributes which were the most impactful as part of the machine learning model were "rf_error_erouter_ip_loss", "rf_error_ipv6pingfailed", and "rebootreason=unknown". These attributes increase the accuracy of the models in determining whether or not the issue is related to internet service / high speed data (HSD).

While the RDK telemetry provided beneficial information for both HSD and video, the VBS data provided the majority of telemetry needed to determine if the customer issue was video-related. The VBS data contains such telemetry as:

- User interactions with their TV via the remote
- Errors displayed on-screen to the customer
- Errors with satisfying a request to the customer
- Etc.

For the billing questions, having data about the customer billing history, specific elements on the bill and differences between those elements over time provided the majority of features needed to satisfy the initial use-case.

4.2. Usefullness

In "Observing home wireless experience through wifi APs" [14], it was shown that many in-home Wi-Fi markers were able to characterize wireless experience. In our work, the definition of telemetry includes error and system logs produced by RDK, which is unique to our use case compared to the literature. We will detail a few examples in this section to give the reader a picture of their utility.

The telemetry includes readings of Wi-Fi signal strength (known as RSSI, for Received Signal Strength Indicator) and channel utilization. RSSI is a negative value (-100,0) and a value below –80 is considered poor signal strength. Therefore, it is clear that this telemetry feature can be useful in identifying devices that have an impacted customer experience. Likewise, channel utilization data can indicate when a given Wi-Fi channel frequency is saturated from too many devices or too much traffic. High utilization is an indication that one may need to change the Wi-Fi channel the router is using for signal transmission.

The RDK logs are another source of events which may impact a customer's service. For example, if a reboot occurs, there are multiple keys that indicate the event has taken place, and in some cases, why the reboot occurred. Unscheduled reboots are generally strong indicators of a customer experience issue.

Moreover, there are system logs when processes on the box restart, which can also impact service, depending on the process. (Such processes are programs running on the thin version of Linux on the gateways). There are other markers like "system_uptime", which gives the time since a last reboot, and "rf_error_ipv6pingfailed", which indicates a count of pings to a fixed IPv6 server address that have failed since the last log. In total, there are over 750 unique keys which appear in RDK and more are added over each release.

5. Platform

5.1. General Architecture

All of the features learned from exploratory data analysis need to be processed, parsed and transformed so they can be passed into the model. During offline training of the model, the researchers perform these tasks but usually at smaller scales. Or, if full-scale, it usually doesn't run constantly. Also in the research environments, the lack of introspection, scaling (depending on the researchers' environment), monitoring and alerting is unacceptable for running a supported product. These are among the goals of an ML platform.

The basic components of the platform, in order of data flow, are:

- Data producer
- Raw data consumer
- Feature engineering
- Model invocation
- Inference storage or action

This typically involves one or more teams and one or more platforms to handle the data. It is common for the platform responsible for polling or collecting data from the CPE to be handled by one team, while a different team is responsible for the ML platform.

When greatly simplified, data collection can be boiled down as seen in Figure 1. Here, some process (usually many parallel processes) is responsible for connecting or listening to the CPE and other devices to collect, format and aggregate telemetry. JSON is typically used for data formatting, though other formats are becoming more common, such as Protocol Buffers and Apache Arrow objects. After the data is formatted, the telemetry collection system will publish the messages to a message bus for consumption from other teams within the organization. For our systems, we use Apache Kafka, as it is stable and scales well.

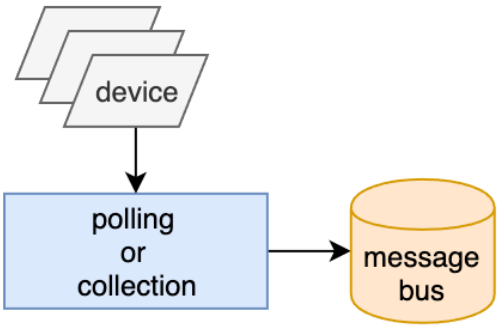

Figure 1 - Telemetry collection flow

Once the data has been published to the message bus, it is decoupled from the data collection platform and available to be used by the ML platform. A simple flow for processing the telemetry events from the message bus is detailed in Figure 2. Here a message bus consumer process will listen to the stream for new messages. Next the message will be parsed, and various transformation and/or normalization steps will be performed on the data to prepare the dataset to be passed into the ML model. Before or after parsing and feature engineering, it is common to store features into a database so they can be later retrieved. This is useful when building aggregate feature sets (such as time windows, feature enrichment, etc.) so the model may be passed a richer and potentially more useful set of features. Once the features are ready, the model is invoked, which produces an inference or prediction. This output is then handled by possibly storing that in another database, distributed filesystem or even published to another message bus to do something with that model output.

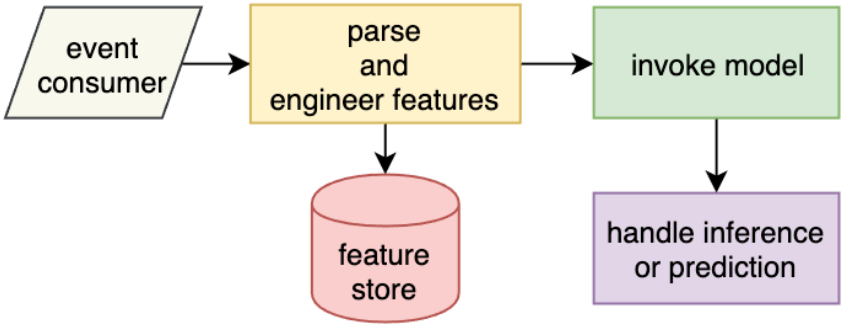

Figure 2 - Feature engineering and model invocation flow

While these two systems are usually logically separated and loosely coupled, thanks to the message bus architecture, they do need to combine as shown in Figure 3. Here the full picture is seen with both the telemetry collection platform and the machine learning platform receiving telemetry.

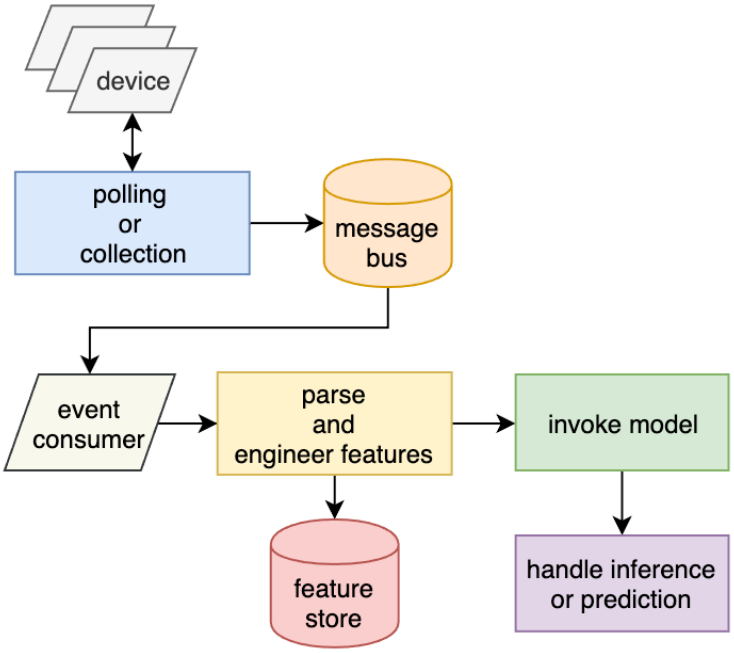

Figure 3 - End-to-end telemetry and ML processing flow

5.2. Implementation

5.2.1. Technology choice

The end architecture contains the same components and flow as shown in Figure 3, but there are differences in how those components look after scaling and with specific technology choices. For the flexibility to run programs of any type, Docker was chosen to contain some of these complex applications. For scalability, a container orchestration layer is required to dynamically scale resources to match the needs of the incoming telemetry – in our case, Kubernetes was chosen here. For a scalable message bus, the choice was Kafka, as previously mentioned. Depending on the rate of operations and type of data in the features, a range of database technologies and types can satisfy that requirement. For example, if you have relational data, a more traditional SQL database such as MySQL or PostgreSQL might be chosen. For a NoSQL database, perhaps Apache Cassandra or MongoDB. For a NoSQL cache layer, CouchDB, Redis, or if SQL, then MemSQL. This is by no means a comprehensive list of database technologies but do represent a few options to investigate, should you have a specific set of requirements for your feature store. For our solution, Redis was chosen primarily for performance reasons and data retention. The research team determined that the largest useful window of data was 24 hours and older historical data beyond that was less impactful. It is also relatively quick to repopulate, as well, in the case of catastrophic cluster issues.

Kubeflow was chosen for model serving not only because it works well with Kubernetes but contains a number of require features as well. More specifically, a sub-component of Kubeflow called Seldon Core was chosen due to its flexible inference graphs, monitoring and A/B testing capabilities. Finally, Python is the language of choice for deploying models due to the fact that the models will require little or no code changes to be deployed. Also, many Python specific packages such as NumPy are very performant and don't have great equivalents in other languages. This also allows the model owner or team less familiar with production operations to maintain and re-deploy their model as needed. Other scientific languages, such as R, have far less library and performance support for writing and running general components.

5.2.2. Architectural differences

The differences between our general architecture in Figure 3 change when we consider Kafka and Kubernetes. Starting with Kafka, it is important to understand a bit about how Kafka works so that we can scale appropriately. Without getting too deep into Kafka's architecture, there are three basic components: the topic (which consumers and producers use), the broker (a node in the Kafka cluster that handles topics) and partitions (a sharded piece of the topic). When we have high volume topics, those usually need to be partitioned out so we can scale throughput horizontally. In Figure 4, the telemetry polling architecture would publish messages to a specific Kafka topic with b_n brokers and P_m partitions.

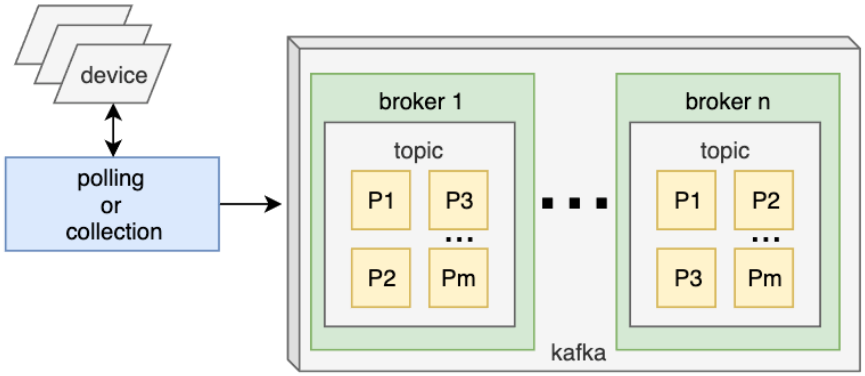

Figure 4 – Message bus parallelism

To calculate n and m, the volume and message size need to be determined. For this example X,Y and Z are just example features. Actual features used in the system are Signal-to-Noise Ratio (SNR), RSSI and others. Take the following row as an example:

Table 1 – Sample telemetry

MAC address	Customer ID	X	Y	Z
aa:bb:cc:dd:ee:fa	123456780	2.156	True	3
aa:bb:cc:dd:ee:fb	123456781	3.211	False	7

If each of these records were represented as individual messages, then they might look like this in JSON.

```
{"mac":"aa:bb:cc:dd:ee:fa", "customerid": "123456780", "x": 2.156, "y": true, "z": 3}
```

However, the data is usually far from optimal, introducing additional parsing overhead or unnecessary extra data. Such as an unnecessary element as shown below "pollresult" or perhaps elements represented as strings instead of their actual datatype.

```
{"pollresult":

      {"mac":"aa:bb:cc:dd:ee:fa", "customerid": "123456780", "x": "2.156", "y":
"True", "z": 3}}
```

5.2.3. Message bus performance

Using this less than perfect data, we can now calculate the volume and size of the messages to determine the correct number of brokers and partitions for the Kafka topic. Using our system as a more realistic example for the message size, the mean message size of 1.4 kilobytes is used. Assuming there are ten million devices being polled every minute and the JSON is raw (uncompressed) on average 1.4kB, and the message above, the throughput would be $\frac{1e7*1400}{1000^2} * \frac{1}{60} = 233.\overline{3}$/MBs. Using 50MB/s per broker as a rule from Dropbox's Kafka Throughput limit post [9], rounding up we would require 5 brokers at 50MB/s each, giving us a reasonable 250MB/s throughput. To determine the number of partitions, multiply the number of brokers by 10, which yields 5 brokers with a single topic of 50 partitions. Of course, this is just an example, and situations are different based on hardware type, network throughput and many other factors.

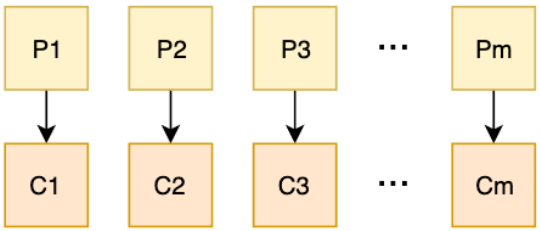

Figure 5 – Message bus consumer scale matching

The reason partition counts per topic are important relates to the performance gain of matching consumer processes or as a multiple of the number of partitions. Here, with 50 partitions, you may need to scale the message bus consumers as shown in Table 2.

Table 2 – Message bus consumer scaling

Consumer count	Multiple	Comment
5	10	If message processing is very fast, single consumer may be able to handle multiple partitions each.
25	2	Many more consumers handling partitions. If message processing is more computationally intensive, each consumer might only handle two partitions.
50	1	A single consumer per partition is required if the code is not only parsing, but performing some action or invoking a ML model inline as shown in Figure 5.

Final thoughts on consumers and messaging or streaming systems: There are many combinations of great open source projects which can be used instead of writing the consumer logic manually. For example, systems like Apache Flink and Apache Beam, as well as offerings from the various cloud providers, can perform many of the functions described here.

5.2.4. Container orchestration

Once the number of consumers has been determined, a scalable, fault-tolerant environment is needed to run the consumers and models. The flexibility of the runtime environment is crucial to operating and maintaining ML models and pipelines, which made Kubernetes and Docker a natural choice. Kubernetes also allows elastic horizontal scaling, to scale up for spikes and scale down for dips in processing.

Zooming in on the general architecture in Figure 3, the message consumption and feature handling, as well as the model invocation components -- all translate to Kubernetes pods. Once the engineer or researcher creates the model, it is pushed to a container repository, then pulled down in Kubernetes to run and monitor, as shown in Figure 6, until there is a change. Kubernetes can scale up and down by adding and removing replicas of a particular pod corresponding to the addition and removal of worker nodes.

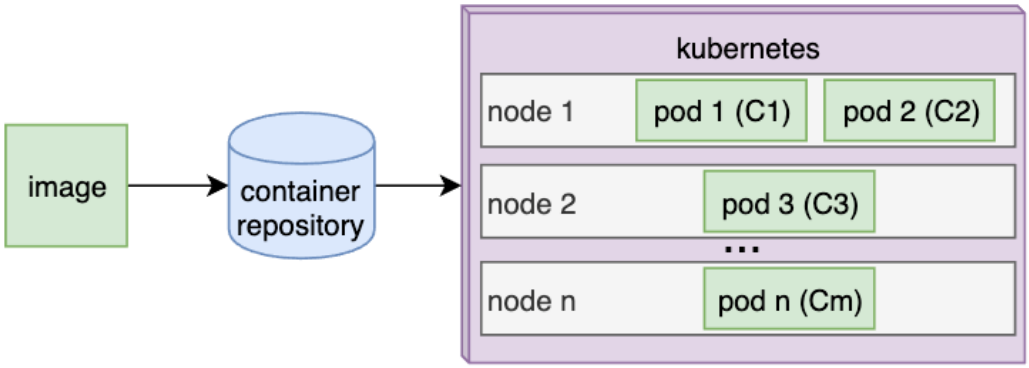

Figure 6 – Container parallelism

A benefit of Kubernetes is that pods of any type can be run together on the same cluster. If the consumer pods for example are written in Java, and the model code is written in Python, then both can run simultaneously on the same worker nodes, thanks to Docker, and shown in Figure 7.

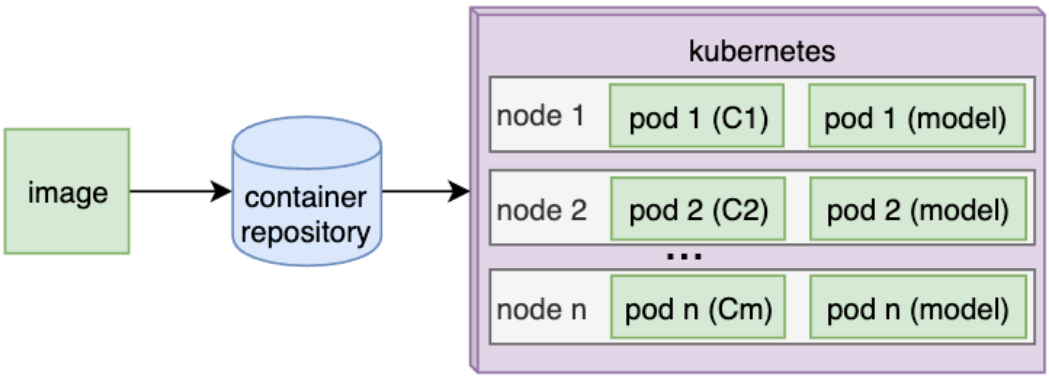

Figure 7 – Container isolation

5.3. Feature engineering performance and guidelines

Just before the model can be invoked, the raw features from the polling architecture require parsing and some manipulation specific to the ML model. Many types of manipulations exist, such as min-max normalization and standardization. For specific ML domains such as Natural Language Processing (NLP) or Deep Neural Networks (DNN), the data may need to be tokenized or one-hot encoded. Three of these normalizations are common, so it is necessary to look at them with the corresponding features and how those features get changed.

5.3.1. Min-Max normalization

To rescale features between 0 and 1, the formula is $x' = \frac{x - \min(x)}{\max(x) - \min(x)}$. However, this requires that the maximum and minimum values of the features are available. Many systems may only contain partial feature sets from which the minimum and maximum value cannot be derived. Note that the term "feature set" is just a series or list of features, and order may be important. Also common is to scale features to a particular range $[a, b]$. For this, the formula is slightly different $x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$. If the features

require these normalizations, the minimum and maximum may be provided, or the telemetry will need to be collected for an acceptable period of time to derive acceptable values.

5.3.2. Sum or counts

Adding or counting features for a particular time window or sample is a common normalization. However, the computational complexity when performing feature engineering at scale here can be deceiving – especially when referring back to the example of ten million devices polled every minute. Take a hypothetical situation where the desired prediction is a usage pattern spike given two features, bytes in and bytes out. Assume that research has found 30 minute windows to be optimal for predicting if there will be a spike. The total feature size for all devices would be $1e7 * 30 = 300e6$ or 300 million. If the features were simply a MAC address, bytes in, bytes out, and timestamp, the JSON representation might look like

```
{"mac": "aa:bb:cc:dd:ee:fa", "bytesin": 1287630, "bytesout": 58360, "timestamp":
"2020-07-01 02:31:05"}
```

The feature set size for all ten million MAC addresses would be $\frac{\sim 103\ bytes}{1024^2} * 10e6\ devices * 30\ minutes$. While this is only ~29.3GB worth of features which is relatively small, the compute time may be large. Consider another example: Python code as pseudo-code. There are a number of details left out, such as actual timestamp calculation, and data structure details:

```
# iterate through current polled messages from the kafka topic for each device

for features in current_features_from_kafka:

 # get history

 history = get_history_for_mac(features['mac'])

 history.append(features)

 # iterate through history

 summed_features = {'bytesin': 0, 'bytesout': 0}

 for history_features in history:

  # expire old items

  if history_features['timestamp'] > thirty_minutes_ago:

   # iterate through features
```



```
    summed_features['bytesin'] += history_features['bytesin']

    summed_features['bytesout'] += history_features['bytesout']
```

Say there are 50,000 messages from Kafka, each having 30 minutes of history, with the computational complexity of $O(n^2)$ (where n is the number of steps), which results in $50000 * 30 = 1{,}500{,}000$ iterations. If all ten million are polled every minute, that is 300 million iterations per minute. All of this requires more replicas to scale or, some simple tuning of the algorithm. If the math is commutative, it can be parallelized out of order, and for a simple count, it is. However, here the values may contain negatives, which means it is non-commutative and must be executed in order, serially. However, these can certainly be parallelized as the only requirement is that each device needs its feature sets to be computed in order. By changing the algorithm to remove iterating through the entire history set for each new message, we can reduce the second loop from 30 iterations to just two operations when storing the summed features as well in a database. There are obviously a few more steps here, such as the back and forth from the database and type conversion (if necessary), but it can all add up -- which can, in some cases, dramatically increase costs. With the improvement, for the 50,000 messages from Kafka we're now only doing 50,000 iterations instead of 1,500,000 and a time complexity of $O(n)$.

```
# iterate through current polled messages from the kafka topic for each device

for features in current_features_from_kafka:

  # get history from feature store

  history = get_history_for_mac(features['mac'])

  summed_features = get_summed_features_for_mac(features['mac'])

  # remove the first (oldest) and decrement from summed_features

  oldest_history = get_expired_feature_from_history(history)

  summed_features['bytesin'] -= oldest_history['bytesin']

  summed_features['bytesout'] -= oldest_history['bytesout']

  # add the current features

  summed_features['bytesin'] += features['bytesin']

  summed_features['bytesout'] += features['bytesout']
```

Said another way, we can look at it as a sum of series, where those series may contain negatives. Let n equal the number of historical feature sets for the series a, and m equal the number of features within each historical feature set a_i. The total sum S_m is the series of all historical feature sets:

$$S_m = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_{ij}$$

Now let c equal the current message being processed with p features. Then the historical features sum the current features:

$$S_m = \sum_{k=0}^{p-1} S_k + c_k$$

If the previous sum of the historical feature sets is known, then let h equal that series, and the original operation can be rewritten more efficiently as a single sum. This does not take into account the subtraction or decrementing of historical feature sets that expired from the desired time window (30 minutes for this example):

$$S_m = \sum_{k=0}^{p-1} h_k + c_k$$

Keep in mind that this assumes the series and elements are the same between series. In real-world data, the elements for each piece of telemetry may be different, resulting in sparse data structures which require different thinking to process efficiently.

5.3.3. Mean

The mean here is a bit more complicated because without iterating through the entire history an actual mean $\frac{1}{n}\sum_{i=1}^{n} a_i$ can't be derived. To get around this and get a "good enough" mean, we turn to a simple moving average, so we don't have to iterate through the entire historical time window of features. With a simple moving average $\frac{1}{n}\sum_{i=0}^{n-1} p_{M-i}$ is used as we have the historical features in the database and the code is similar to the sum example.

5.3.4. Other operations

For operations such as standard deviation, minimum or maximum values are more complicated to process given a single value. There are potential ways to calculate these things given more values, such as minimum or maximum, if you know the top or bottom 5 for a particular feature. This requires keeping those lists and can be cumbersome -- which may defeat the purpose of computational efficiencies, unless the historical time window or number of messages therein is large.

6. Model Serving

The last component of the system is the model, and how it gets exposed to users or other parts of the organization. This is perhaps the most straightforward piece of the system, not considering what happens to the output (prediction or inference) from the model. The simplicity comes from a project called Seldon Core which allows almost any type of ML model to be invoked with Python and served from Kubernetes. In Figure 7, not only is the model potentially running alongside the feature engineering on the same node, but in separate containers, it is also able to scale horizontally by simply changing the number of replicas. This can also be performed automatically with elastic scaling. All the major cloud providers support some kind of model serving and each has their own benefits and drawbacks depending on the situation. Running your own Kubernetes allows you to also run Prometheus or tie it into an existing monitoring solution within the organization, such as to leverage metrics and alarming for the operations support side of these platforms.

6.1. Requirements

Given the large number of components in the model, a system which supported core requirements such as multi-armed bandits, A/B testing and advanced inference graphs was required. Seldon Core was chosen as it satisfied all of these requirements and integrated well with Kubernetes and production monitoring systems. Seldon can support a user-defined inference graph, which allows custom configuration and combinations of models and components. It also allows for percentages of traffic to be handled by each, thus allowing for more complicated implementations such as A/B testing. As part of the user request flow to the model, it was important to ensure that customer HTTP sessions are sticky to a particular A/B model. In order to accomplish this, a custom Seldon router, which takes a seed value can be written to route requests to the same components in the inference graph depending on values in the payload (such as a customer identifier.

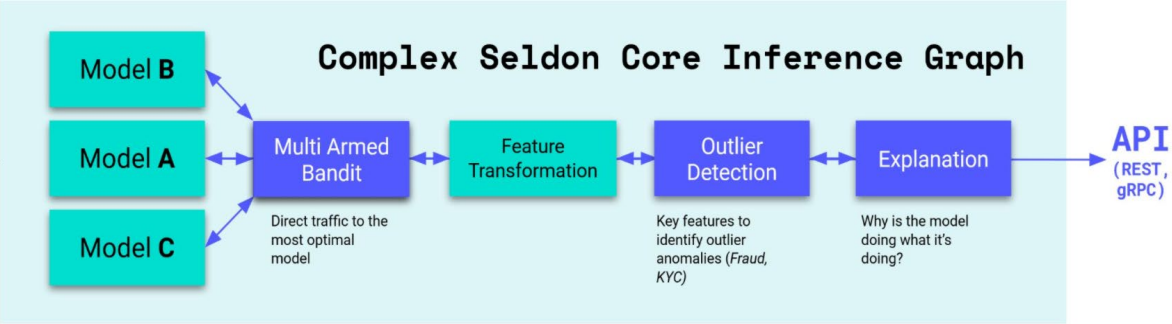

Figure 8 – Inference graph example

6.2. End-to-end flow

While the entire flow is shown in Figure 3, there is a critical piece missing. Figure 9 depicts the end-to-end flow, including customer interaction with the system to request predictions from the model.

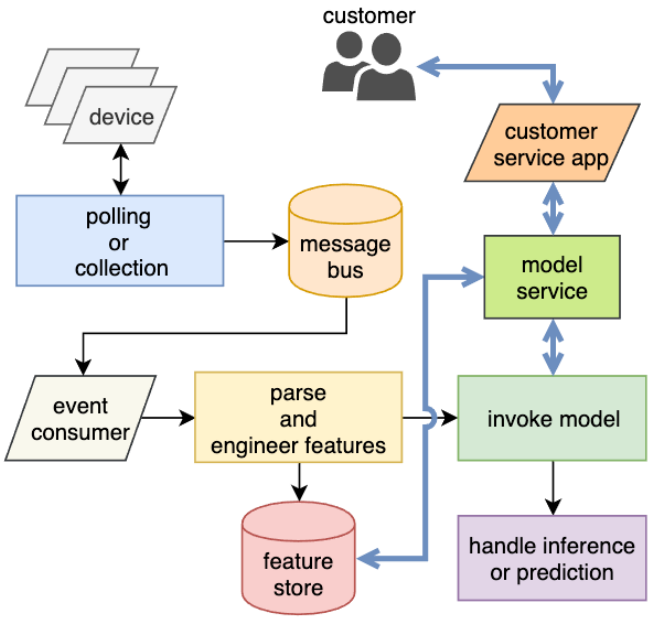

Figure 9 – End-to-end platform flow

In the second flow with the thick blue lines, the customer interacts with the customer service application on their device or web browser. This invokes the model service and returns the prediction back to the application, then the user, about whether or not it believes they are experiencing an issue based on the telemetry and output from the application. If there is an issue, it attempts to specify a recommended fix action, or, as a last resort, refer the customer to customer service. This system has been successful at handling millions of customer requests quickly and accurately, improving the customer experience.

7. Scale

In our customer experience ML platform, we're generally handling 100k+ messages per second and executing 300-500k+ operations per second to our Redis clusters. As previously mentioned, this is accomplished with horizontal scaling from Kubernetes. In many of our platforms, we're handling more than just the RDK datasets, as our models require engineered features from many varying datasets to produce the unique outputs to improve the customer experience.

Our Kubernetes clusters generally run with 10 or more nodes at 16 CPU cores per node and 128GB RAM. The Redis cluster nodes have far fewer CPU cores (4) as the Redis process is single-threaded, yet they have much more memory, generally 256GB RAM. Our Redis nodes as well as the Kafka broker nodes have secondary 1TB drives for disk-based persistence, replication and retention.

8. Model

Our goal is to prompt a user to troubleshoot one of their services in order to increase engagement and troubleshooting inside the chat bot. Thus, our data is generated based on both customer feedback and the decision to show a prompt to a user. In this way, the model will impact the data generating process, which is different from standard classification or regression use cases, where the data generation is assumed to

be independent of the model's output. For example, in image classification, if a model incorrectly labels a picture of a cat as a dog, there is no impact to the ground-truth label—that the photo is of a cat. In our case, we will have feedback based on the action we select, but we have no information on if the model took a different action. Our problem can be formalized in the framework of contextual bandits.

Contextual bandits algorithms extend the traditional multi-armed bandits problems by giving the agent a state to aid in decision making. Each data point can be thought of as a "round" of a game, where the state is the features of the model ("agent/policy") , the action is chosen by the agent, and the reward is the label. Contextual bandits problems are a special case of Reinforcement Learning, where the length of a round is 1. The goal of the agent is the learn to play the game in order to maximize the expected reward when following the agent's action recommendations.

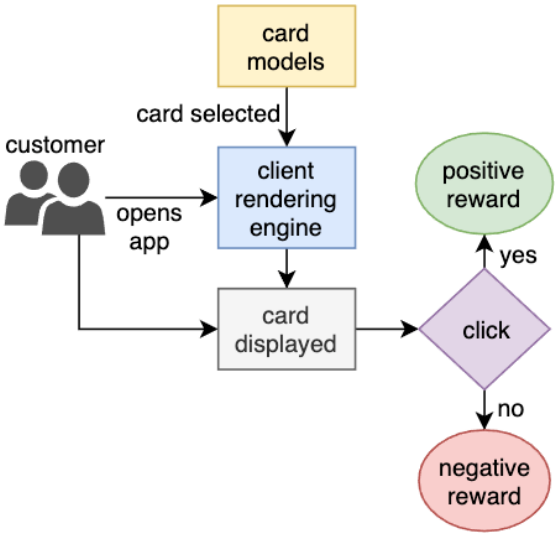

Figure 10 – Model flow

One drawback to reinforcement learning and contextual bandits in industry is that the algorithms are traditionally trained online. In the case of reinforcement learning, agents typically have access to a simulator of the environment (i.e. a game) and they learn as they play (c.f. "Playing Atari with Deep Reinforcement Learning" [17]). For contextual bandits algorithms, they often need to be deployed into production and learn as they serve customers [15], [16]. This is because the data distribution changes as the agent learns, as the agent will select new actions based on new information. This is different from the static training set found in classic supervised learning. Because of this, a suite of theoretical techniques has been developed to evaluate and train agents offline, namely "off-policy" techniques [18], [19]. Off-policy techniques have been employed in a variety of industrial applications successfully, including [20], [21]. This approach utilizes data from a production logging policy with sufficient randomization for training and evaluation of offline policies.

One of the large benefits of off-policy evaluation is the ability to estimate the true online performance of a contextual bandits model without deploying it into an A/B test. This allows the researcher to rapidly prototype many candidate models and choose the best model for an A/B test which impacts customers.

With that said, our model is a linear model which predicts the reward for each action (prompt) that we may show the user given the state, i.e. telemetry features defined above. After predicting the expected

reward for each action, we take the action with the highest reward and output that action to the production system. We train the model using various off-policy approaches [22] and select the model which performs the best on the off-policy evaluations of our metrics for production. Crafting reward functions is an art more than a science, and generally are designed by experts to maximize not only the business metric of interest but also downstream metrics. In our case, our reward function is a proxy for the click-through rate of the prompt, where we have a negative reward if a card was shown and not clicked. We scaled reward functions to be between [-1,1], as it makes the training process more stable through smaller gradient updates.

All analysis was done in Jupyter Notebooks, including training. Models were defined and trained using Keras, a popular abstraction of neural network operations that sits on top of libraries like Tensorflow. In production, the model is invoked to select a card for the user upon application start.

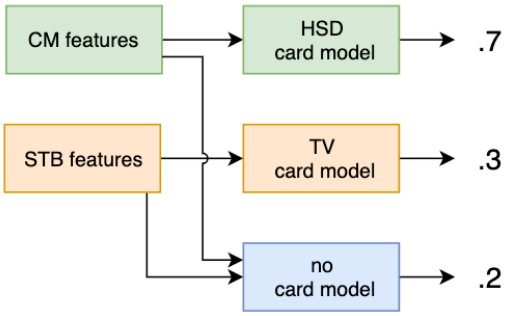

Figure 11 – Model architecture

8.1. Feature Engineering

The telemetry we utilize comes in several forms: snapshots of physical characteristics of the Wi-Fi signal; device performance indicators usually expressed as a percent; and application logs which count the occurrences of different system messages, including error codes. We pass each of these types of features through a different pipeline to create appropriate features. All of our features are aggregations over window of time *t,* where *it* depends on the feature type.

For continuous features, we utilize statistical aggregations over the time window. Many of the continuous features are related to devices connected to the gateway, and so we will have multiple devices for a given customer. We take an aggregation of statistics calculated on each device as our final feature. For example, consider the RSSI: we calculate the mean, median, the standard deviation, and other statistics for each device, and then we aggregate these statistics again for a single value for each customer account. Time aggregations are used so that the model has is able to learn a sense of what's "normal" and can identify deviations from those norms. More significant deviations could imply impact to a customer.

For count valued features, we base our approach on common natural language processing (NLP) techniques. The counts of the more common errors/logs over a time window can indicate the intensity of a problem, and the existence of rare keys can indicate a complete service impairment. Thus, common NLP techniques are applicable to our use case and provide a framework for creating features (embeddings) from which a model may learn. Each error code or system message indicator is treated as a "word" and

passes this context to our feature pipeline as a "bag of words". The data is then transformed via a pre-trained TF-IDF pipeline before using it in the model.

Our feature processing pipelines use classes from sci-kit learn. For example, all features are normalized before training using preprocessing classes. Continuous features are standardized and (sparse) count features are scaled by their maximum absolute value. In addition, the TF-IDF embeddings are implemented using the corresponding classes found in scikit-learn for this purpose.

9. Results

By utilizing the machine learning model, we achieved a 40% improvement in customer engagement, as measured by the abandon rate of sessions (when a customer opens the application and leaves without clicking or asking a question). This was measured through an A/B test where all customers selected to get a card were split into two groups: the treatment group who were shown the card and a control group where we held back the card. This business impact was accompanied by a measurable increase (~42%) in the click-through rate of cards when compared to simple card display rules based on telemetry cutoffs. This improvement was calculated by comparing each approach to a baseline over their respective time periods in production to control for temporal differences. The final number is the relative difference between the two statistics after controlling for the baseline.

Using the general architecture depicted in Figure 8, with simple horizontal scaling, the platform can process very large amounts of incoming telemetry data. When converted to feature sets, the production systems are handling 100k transformations per second. This is using feature windows that vary between 6 and 24 hours based on the dataset and requirements for the model. This scaling is nearly seamless if the code is written with good microservice design principles in mind, and the ease of increasing replicas for an application is handled. Kubernetes also supports elastic scaling based on certain metrics, which is useful for processing spikes or bursts in traffic.

Scale aside, researchers and ML engineers have the ability to deploy models and feature pipelines as needed. This reduces code rewrite from POC or research code to production code as the production systems support many languages, ML model types and complex pipelines. Additionally, troubleshooting and operations (MLOps) is greatly aided by Prometheus integration with thresholding from Grafana and other tools to ensure problematic components are dealt with quickly. System availability is high due to the redundant nature of Kubernetes and requiring all components have a minimum of two replicas and dual components when they lie in the critical path.

10. Looking Ahead

These sorts of open source platforms and architectures are simply the beginning of a sweeping change coming to not only telecommunications, but every other industry. Big telcos certainly have an advantage now, given the vast datasets being collected now or in the future. While the physical properties of components and systems are well understood, there is always room to identify new correlations in failure or impairment types. With new hardware, more optical components and higher speeds -- and the fact that connectivity is so vital to everyone -- this shift to self-identifying and self-healing networks will become more critical. While ML models are not specifically required for every problem, the collection and analysis platforms are, because the data is required to work towards more data-driven decision making. Also, in the future, the platform output itself will be fed through various anomaly detection model types to determine if system performance has deviated sufficiently to warrant human interaction. Platform failures and other behaviors can be modeled to increase self-healing, predictive scaling (as opposed to reactive) and automated root-cause analysis -- down to the specific component.

These platforms will need to adapt in the coming years, to leverage FPGA or GPU hardware to handle increasingly more complex analysis and ML workflows. Many companies are already seeing the benefits from GPU, though TPUs and other hardware will be available to further drive down these large costs.

11. Conclusion

Again, the core problem here is that customer service and many other applications may require ML to make a measurable impact in customer experience. As we have shown, not only were many open source technologies instrumental in building the solution, they were not the only important pieces. These systems may need to scale which will require deep dives into some of the core components that get executed many times over, and those should be optimized as much as possible. While choosing Python (the same language as is common for research) initially proved troublesome, performance has increased dramatically and new avenues to optimize these functions, such as GPU offload have come a long way.

When thinking about the model, effort is required to determine if ML is necessary at all and how it can make a positive impact on the problem. For this problem, a contextual bandits approach was chosen but that does not mean is it the correct solution for other similar problems.

By now, after reviewing the problem, solution and challenges to solving this problem at large scale, it is hoped that this information is not only helpful but the start of a change in thinking about how problems will be solved going forward into the next decade. ML is not beginning but continuing to alter many parts of our organizations and this will only accelerate. The information presented here may be somewhat specific to telco telemetry data, but in fact it can be applied to most any similar dataset.

Our customers are the core of what we do and drive us to strive to do better in terms of not only providing services but help when those services become impaired. This solution and others not only aids in our graceful and accurate handling of these, but the hope is for all of us to continue to think about these problems from the customers perspective.

Abbreviations

CM	cable modem
CMTS	cable modem termination system
CPE	customer premise equipment
DNN	deep neural network
FPGA	field programmable gate array
GPU	graphics processing unit
HSD	high speed data (internet service)
IoT	internet of Things
JSON	JavaScript object notation
OS	operating system
MAC	media access control address
ML	machine learning
MLOps	machine learning operations

NLP	natural language processing
POC	proof of concept
POP	point of presence
RDK	Reference Design Kit
RSSI	received signal strength indicator
STB	set-top box
TF-IDF	term frequency, inverse document frequency
TPU	tensor processing unit
VBS	video backend services

Bibliography & References

[1] *Data Over Cable Service Interface Specification Proactive Network Maintenance*, PNM Best Practices Primer: HFC Networks (DOCSIS® 3.1) CM-GL-PNM-3.1-V01-200506, 05/06/2020, https://www.cablelabs.com/specifications/CM-GL-PNM-3.1

[2] *What is Customer Experience*, Blake Morgan; Apr 20, 2017, https://www.forbes.com/sites/blakemorgan/2017/04/20/what-is-customer-experience-2/#8b5958170c2b

[3] Reference Design Kit (RDK), https://rdkcentral.com

[4] *Protocol Buffers*, https://developers.google.com/protocol-buffers

[5] *Apache Arrow*, https://arrow.apache.org

[6] *Apache Kafka*, https://kafka.apache.org

[7] *Docker, Inc*. https://www.docker.com

[8] Kubernetes, Linux Foundation, https://kubernetes.io

[9] Peng Kang, Finding Kafka's throughput limit in Dropbox infrastructure, Jan 30, 2019, https://dropbox.tech/infrastructure/finding-kafkas-throughput-limit-in-dropbox-infrastructure#:~:text=This%20result%20provides%20guidelines%20for,throughput%20of%20future%20use%20cases.

[10] *Apache Flink*, https://flink.apache.org

[11] *Apache Beam*, https://beam.apache.org

[12] *Seldon Core,* Seldon Technologies Ltd, https://docs.seldon.io/projects/seldon-core/en/v1.1.0/

[13] *Feature Scaling*, https://en.wikipedia.org/wiki/Feature_scaling

[14] CITE-WIFI *Observing home wireless expe- rience through wifi aps*. Ashish Patro, Srinivas Govindan, and Suman Banerjee; In Proceedings of the 19th Annual International Conference on Mobile Computing & Networking, MobiCom '13, page 339–350, New York, NY, USA, 2013. Association for Computing Machinery.

[15] CITEA *The nonstochastic multiarmed bandit problem.* Auer, Peter, Bianchi, Nicol`o C., Freund, Yoav, and Schapire, Robert E.; SIAM Journal on Computing, 32(1):48–77, 2002.

[16] CITEB *The epoch-greedy algorithm for multi-armed bandits with side information*, Langford, John and Zhang, Tong; In Advances in Neural Information Processing Systems 20, pp. 817–824, 2008.

[17] CITEC *Playing Atari with Deep Reinforcement Learning*, Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller; https://arxiv.org/abs/1312.5602

[18] CITED *Learning from Logged Implicit Exploration Data*, Alex Strehl, John Langford, Sham Kakade; 14 Jun 2010, https://arxiv.org/pdf/1003.0120.pdf
[19] CITEE *Batch Learning from Logged Bandit Feedback through Counterfactual Risk Minimization*, Adith Swaminathan, Thorsten Joachims; 2015 https://www.cs.cornell.edu/people/tj/publications/swaminathan_joachims_15c.pdf
[20] CITEF *A Contextual-Bandit Approach to Personalized News Article Recommendation*, Lihong Li, Wei Chu, John Langford, Robert E. Schapire; 1 Mar 2012, https://arxiv.org/pdf/1003.0146.pdf
[21] CITEG *Top-K Off-Policy Correction for a REINFORCE Recommender System*, Minmin Chen, Alex Beutel, Paul Covington, Sagar Jain, Francois Belletti, Ed H. Chi., 6 Dec 2018 https://arxiv.org/pdf/1812.02353.pdf
[22] CITEH *Tutorial on Counterfactual Evaluation and Learning for Search, Recommendation and Ad Placement*, Thorsten Joachims, Adith Swaminathan; SIGIR 2016, 17.07.2016, http://www.cs.cornell.edu/~adith/CfactSIGIR2016/

