Apache Kafka for Real-time Reinforcement Learning

ST446 Distributed Computing for Big Data

Student 10173

10 May 2021

Contents

1. Introduction and Motivation

Reinforcement learning is well known for its great achievements in playing games such as Chess, Go, or Atari. These algorithms are typically trained in a simulated environment (such as the games themselves).

In real life, reinforcement learning algorithms are not yet very widely spread. This is not only due to the relative complexity of the algorithms themselves, but also because they require a quite sophisticated technical setup - the overhead incurred for the deployment of a machine learning model is also known as the *hidden technical debt*, which is particularly high for RL algorithms. Of course, ethical aspects also play an important role. In many cases, it is just not appropriate to learn by interacting with the real world (for example in health care).

Nevertheless, real-world reinforcement learning has a huge potential, especially in cases where batch learning is simply not good or fast enough. For example, people's preferences can change quite quickly. A news article that was popular in the morning may not be relevant any more in the afternoon if there was a major breaking news story in the meantime. Such changing circumstances are also known as *concept drift*, and they may occur as rapidly as just described or gradually over a period of days or weeks. Reinforcement learning algorithms, which are typically online algorithms, can adapt to such changes - even (almost) instantly if necessary.

When it comes to streaming data and real-time analytics, there is one technology that stands out as the most widespread in practice: *Apache Kafka*. It is already in use at many organizations because it serves countless different use cases. It is therefore only natural to wonder whether Kafka can also be used as the "nervous system" of a real-time reinforcement learning system. If so, it could significantly reduce the hidden technical debt of real-world RL algorithms.

The objective of this project is to design and implement a multi-armed bandit based recommendation system that uses Kafka as its central streaming component that allows online learning from feedback in (almost) real-time.

My implementation of such a system shows that Kafka is indeed a great choice for real-world reinforcement learning problems. In particular, I demonstrate that it can solve the multi-armed bandit problem and lead to vastly better recommendations than random selection. It can also deal with concept drift, which often occurs in practice.

Accompanying this report is a Jupyter notebook (*Project Notebook.ipynb*) which comprises a documentation of all the technical aspects of this project such as the cluster setup and any shell commands. The notebook also includes screenshots which show that my implementation works as intended. It also includes the results and plots I present here. My suggestion is to first read this report, then the Jupyter notebook, and finally the Python files.

2. Problem Description

My objective for this project is to design, implement and test a distributed, Kafka-based streaming architecture that suits the requirements of real-time reinforcement learning. For this reason, I first give a quick overview of reinforcement learning and the particular use case I consider.

2.1 Reinforcement Learning and Multi-Armed Bandits

The problem I am trying to solve is a simple reinforcement learning problem. In particular, I consider a multi-armed bandit (MAB) scenario in which different arms are associated with different expected rewards, i.e. on average some arms are better than others and the goal of the agent (or bandit) is to find the best arm. This is an online learning scenario as it is based on sequential decision making.

In each round $t = 1, ..., T$ (Lattimore and Szepesvari, 2019),

1. the agent chooses arm K_t according to some explore-exploit action selection (see explanation below)
2. the agent observes reward R_t and updates its reward estimate for arm k

A common theme in reinforcement learning is the tradeoff between *exploration* and *exploitation*. In order to find the best arm, the agent has to explore them. At the same time, however, the main goal is to maximize the total reward over T rounds, and in that sense it is better to exploit the (what is thought to be) best arm. Neither pure exploration nor exploitation generally leads to good results.

A simple yet effective way to balance exploration and exploitation is to use an ϵ-greedy action selection (also called a *policy*). It means that in each round, the agent chooses the best arm with probability $1 - \epsilon$ and a random arm with probability ϵ. In order to determine what the best arm is, the agent must keep track of the average reward $Q(K)$ of each arm, which is computed as shown below. With some abuse of notation, Q may therefore be considered the "parameters" of the ϵ-greedy policy.

$$Q_{t+1}(K) \leftarrow Q_t(K) + \frac{1}{N_t(K)}(R_t - Q_t(K))$$

where Q and N are k-dimensional vectors of the reward estimates and arm counts, respectively (Sutton and Barto, 2018).

2.2 News Article Recommendation

More specifically, I consider the use case of news article recommendations. This is one of the most commonly utilized scenarios with respect to multi-armed bandits (Li, Langford and Schapire, 2012). In fact, my problem setup is based on real click log data collect by Yahoo in 2009 (more information about the *Yahoo! Front Page Today Module User Click Log Dataset* can be found **here**).

In this dataset, there are 20 different news articles which correspond to 20 different arms of a multi-armed bandit. The news articles were randomly presented to users (i.e. visitors of the website) and the rewards were recorded. A click means a reward of 1 while no click means a reward of zero.

Both articles and users are additionally associated with a feature vector, so this data can also be used for more sophisticated contextual bandit algorithms. I will not make use of these features, however, and instead only a consider the multi-armed bandit problem.

Simulated Online Scenario. I use this dataset to simulate an online learning scenario. This is necessary since learning from a logged dataset would otherwise have to be done offline, which is not the objective of this project. Rather, the goal is to learn in real-time, i.e. online. I simulate the online scenario in the following way.

First, I compute the average observed reward of each news article from a subset of the Yahoo Click Log dataset (10,000 rows) and I assume these averages represent the arms' ground truth click probabilities. In other words, the reward for each arm is assumed to be a Bernoulli random variable with probability equal to this ground truth. The plot below shows these 20 ground truth click probabilities and the average reward (in red).

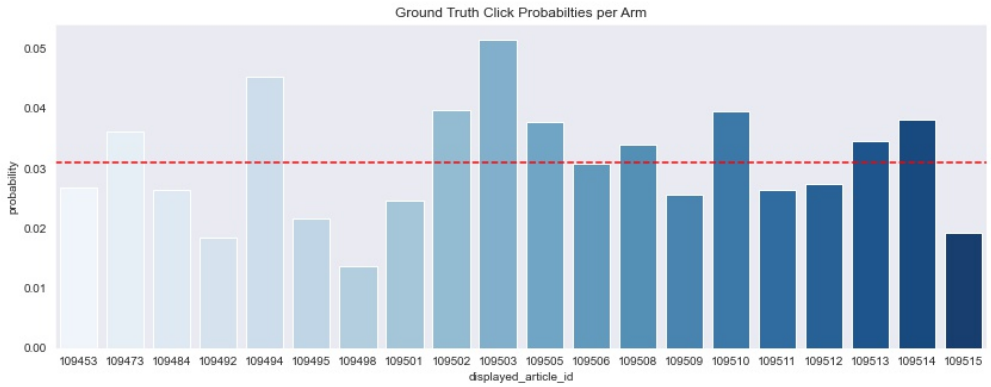

Figure 1: Ground Truth Click Probabilities

As can be seen, some articles have a higher click probability (i.e. a higher expected probability of a click) than others, ranging from approximately 1 to 5 percent. I embed this ground truth into my *application.py* file, which is responsible for simulating the training data, including the rewards (see chapter 3 for more details). The goal of the agent is to present the article with the highest click probability to each user.

3. Solution Architecture

It is relatively easy to solve this problem with an agent that selects actions and learns from its decisions. In practice, the situation is more complicated. First of all, it is typically impractical to use the same model (agent) for both learning and inference (i.e. making recommendations). This is because unlike in a simulation, training data does not necessarily arrive strictly sequentially. Second, a single model may not be able to handle big data in terms of both volume (millions of events may happen) and velocity (many events may occur within milliseconds). Third, any recommendations must be made or sent to wherever the user is, and any rewards must be collected and aggregated, which comes with additional challenges such as latency or even delayed feedback.

The trigger for a recommendation may be a person entering the news website, or a smartphone user opening a hypothetical news application on her device. In fact, the latter scenario will be particularly useful in describing my solution architecture.

I consider a hypothetical scenario in which multiple applications ("news apps") require recommendations. Each app runs on a separate device that is connected to the Kafka cluster, but devices are not directly connected to each other.

Of course, each device does not have access to the data from other devices. What should be clear from this is that learning can only occur in a centralized fashion since the data from all applications must be aggregated. It would be impractical to distribute all the data to all devices.

Consider the following procedure. An interaction starts when a user opens the news app. She is then shown one of the 20 news articles, which is the article recommended by the ϵ-greedy policy. A binary reward is observed and the training data, which simply consists of the (K, R) tuple, is sent to the online learner.

Where exactly the decision making takes place and how the data involved is exchanged will be discussed in more detail in chapter 3.2.

3.1 Requirements

Before looking into concrete solution architectures for this problem, however, it is important to first outline which requirements any solution should satisfy. The following five are especially important.

- **low latency**: a user should not have to wait for her recommendation
- **high throughput:** there will potentially be many users at the same time
- **reliability**: fault-tolerance and high availability are generally important for big data applications, i.e. there should be no downtime
- **persistent storage**: despite stream learning, persistence of the data is still important; sometimes historic data can be reused, for example for offline training
- **always up to date**: recommendations should always be made based on the latest training data; that means updates should occur at a high frequency

3.2 Two Possible Solutions

As already mentioned, my objective is to solve the multi-armed bandit problem introduced above in a distributed fashion, using Kafka as the central nervous system. My setup is based on a separation between hypothetical "news apps" and an "online learner", which is an online learning algorithm. All the data involved should be streamed via Kafka.

There are countless different possible architectures and none is strictly better than all others. In fact, any design choices should depend on the use case at hand. The two possibilities I would like to highlight here are **central policy serving** and **policy deployment at the edge**.

Note that I use the term "policy" here because it is more descriptive in the reinforcement learning context whereas in a more general machine learning context, we may call it "model". In fact, a policy is often defined by a model and as a result, any policy updates are typically just updates to the model's parameters (see section 3.4 Implementation Details for more details on how this translates to the multi-armed bandit setting).

Option 1: Central Policy Serving

One possibility is to place the production policy (i.e. the policy that makes recommendations) on a dedicated policy server. The server directly receives requests from applications (e.g. via HTTP) and responds with recommendations. Training is strictly separated from inference and can be done either fully online (one by one), in mini-batches, or by retraining (i.e. essentially offline). The training data is streamed through Kafka and updates to the production policy can be made at any time by simply updating the parameters of the policy in production, which can also be done via Kafka. It is possible to perform such updates in this scenario at a quite high frequency, say every few minutes or even more often.

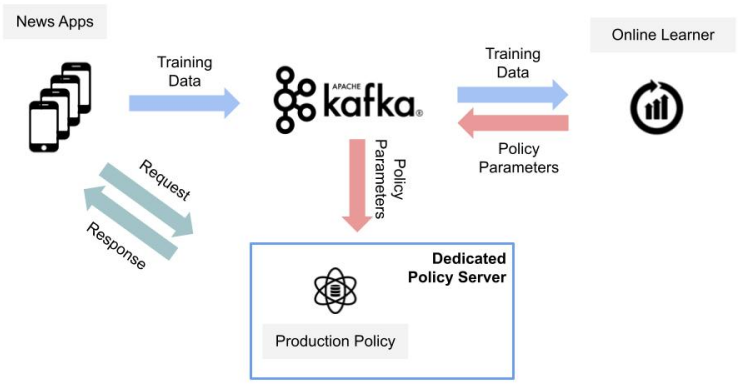

Figure 2: Central Policy Serving Architecture

Option 2: Policy Deployment at the Edge

The second option follows the idea of *edge computing*. That is, the policy itself is embedded in the application, and inference (i.e. predictions/recommendations) occurs locally on the device. Training is still performed centrally using data streamed via Kafka (again either fully online, in mini-batches, or offline). However, the main difference to central policy serving is that now there is not a single policy that must be updated, but one policy at each device. Of course, in order to push any policy updates to the edge, Kafka can be used.

Figure 3: Policy Deployment at the Edge Architecture

Both methods have their pros and cons. However, I would like to emphasize two key advantages of the edge computing setup.

- **lower latency**: inference happens on the device, no HTTP request/response is required
- **works offline**: even if the policy parameters cannot be updated, the current ones are still available and can be used for making predictions (i.e. recommendations)

For these reasons, I choose to implement the edge computing architecture. With the rise of IoT (internet of

things), such architectures will become more and more important and ubiquitous. Indeed Kafka is a central component in many IoT systems.

3.4 Implementation Details

The **online learner** (see *online_learner.py*) pushes its most recent policy parameters to the Kafka topic "*policy*" at a frequency far less than a second. As already mentioned, a policy is often characterized by a model and its set of weights (the parameters), which is why it typically suffices to update those parameters. Note that in the multi-armed bandit setting, only a vector containing the running averages of the rewards of each arm is required. With some abuse of notation, I still call the reward estimates"policy parameters" in order to keep the description generic.

When an **application** (see *application.py*) starts, it pulls the latest parameters from the Kafka *policy* stream and uses them to make an ϵ-greedy recommendation. Once the recommendation has been made, a binary reward is observed (which is simulated based on the ground truth click probabilities) and the application pushes a new tuple (K, R) as training data to the Kafka topic "*rl-stream*".

The online learner, in turn, also constantly reads the new training data in order to update the policy parameters, which are then pushed to the "policy" stream, as already mentioned. It is essential to understand that both the application and the online learner are Kafka consumers and producers at the same time:

- The online learner consumes training data and produces policy parameters.
- Each application consumes the policy parameters and produces training data.

Also note that this architecture is quite general and flexible in that it generally does not matter what the model or the parameters look like. Even the ϵ-greedy policy itself can be easily changed to any other policy, for example an upper confidence bound (UCB) policy.

Simulating Multiple Applications. In reality, there would be multiple applications operating at the same time. In order to simulate this scenario, I use a while loop within my *application.py* file. The loop ensures that the following steps repeat indefinitely:

1. read parameters from the *policy* stream
2. make an ϵ-greedy decision
3. observe a reward
4. push the new training observation to the stream *training*

The four steps essentially comprise what would happen once within a single app, but by repeating them indefinitely, it is as if there were multiple apps (see Limitations at the end of the report).

4. Results

The baseline to beat is the observed average reward calculated from the Yahoo data, which is 0.031. It is important to remark that the data was collected by randomly choosing arms (i.e. displaying articles). The goal, of course, is to improve upon this baseline.

4.1 Stationary Problem

I first consider a stationary problem in which the ground truth click probabilities do not change throughout the experiment. The updates to the reward estimates, as already described, are computed as follows.

$$Q(k) = Q(k) + \frac{1}{N(k)}(R - Q(k))$$

The plot below shows the average reward obtained by the multi-armed bandit (i.e. the average reward up to a certain timestep). At first, the agent does not have an accurate estimate for any of its arms. In this phase, it performs quite poorly and in fact does worse than the baseline - this is also known as the *cold-start problem*.

After only a short period of choosing low-probability arms, however, the agent soon learns to recommend articles which have a higher probability than the baseline, even though it may not immediately find the best arm. The overall average reward collected after around 11,000 timesteps is approximately 0.045, which is a 50 percent improvement in clicks over the baseline of 0.031. Note that running the whole experiment for the approximately 11,000 timesteps took only around 30 seconds.

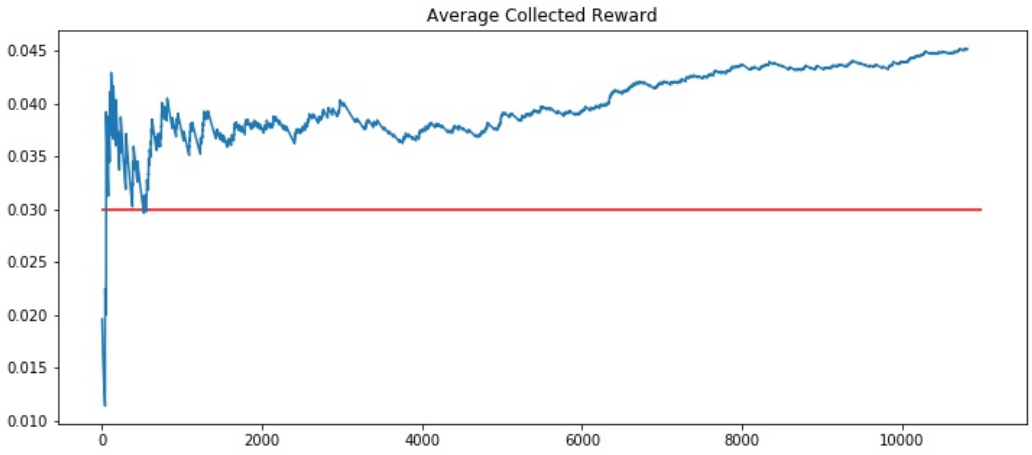

Figure 4: Average Collected Reward (Stationary Problem)

In addition, it is also insightful to examine the history of the reward estimates per arm (plot below). We can see that during the first 5,000 timesteps, the agent chooses sub-optimal arms most of the times (i.e. it behaves greedily with respect to a non-optimal arm). This may not be too bad, however, since a not-optimal arm may still be much better than the baseline.

After around 5,000 timesteps, we can see that arm #9 has the highest estimated reward (turquoise line) and is therefore chosen most often by the agent thereafter. Indeed, the estimate converges to around 0.05, which is also the ground truth click probability for this arm.

There are also other interesting patterns. For example, the many smaller and larger "jumps" in the estimates indicate how rare clicks is for most arms. Some arms receive a first positive reward (i.e. a click) only after many thousands of timesteps. This is also due to the ϵ-greedy policy, which simply picks the most promising arm most of the time. This is often what we want, since not too much exploration should be wasted on obviously bad arms.

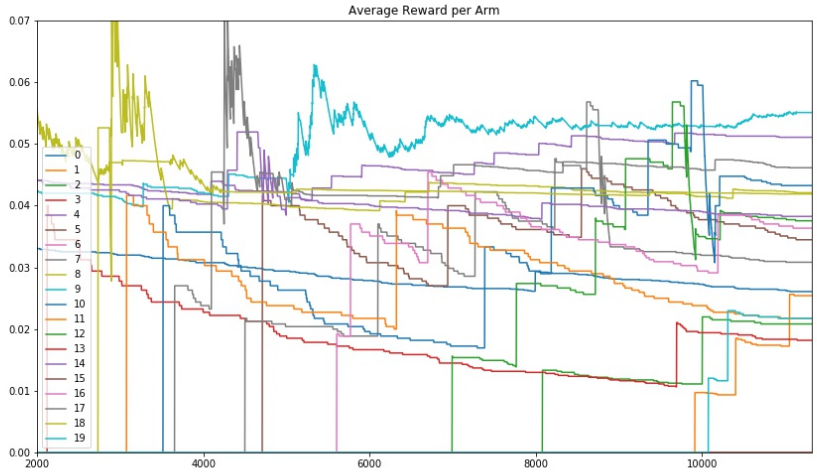

Figure 5: Estimated Rewards per Arm

4.2 Non-Stationary Problem (Concept Drift)

I also consider a non-stationary problem in which after 8,000 timesteps, the click probabilities for all articles plummet by a factor of 5, while one article's click probability jumps to 0.15 In contrast to the stationary problem, this requires a different update function for the reward estimates, i.e. we need a fixed step size.

$$Q_{t+1}(K) = Q_t(K) + \alpha(R_t - Q_t(K))$$

This is also known an *exponential recency-weighted average.*

As can be seen in the plot below, after 8,000 time steps, the reward begins to drop gradually because the algorithm is still recommending the article that was most popular before the sudden change. It takes around 2,000 timesteps for the agent to figure out that there is a new optimal arm now. However, once it finds out, the average reward rises quickly.

Figure 6: Average Collected Reward (Non-Stationary Problem)

5. Conclusion

Real-time reinforcement learning algorithms have enormous potential. Due to the high hidden technical debt, however, they are not yet very widely used in practice.

In this project, I show that Kafka is well suited as the nervous system for real-time reinforcement learning. I describe two solution architectures that are suitable to solve the multi-armed bandit problem, which has countless applications ranging from health care to movie recommendations. One architecture relies on a dedicated policy server, the other is based on the concept of edge computing.

With my implementation of the latter architecture, I demonstrate that Kafka is indeed a great choice for a low-latency recommender system in which recommendations must be made on the most up-to-date information. I implement a simple online learning algorithm that performs policy updates based on aggregated training data from multiple (simulated) applications. Those updates occur almost instantaneously as new training data arrives (at least several times per second). I show that the online learning algorithm can also handle concept drift, an issue that occurs quite often in practice but that traditional batch machine learning models cannot easily cope with.

Of course, the frequency of the policy updates can also be lower than several times per second, if necessary. In some circumstances, it may be more appropriate to perform updates only after some time interval (say, one hour) or after a certain number of new training observations are observed (say, 1000).

My implemented architecture also meets all requirements that I outlined in chapter 2.3. I already mentioned the **low latency** and that the policy is always **up to date**. In addition, Kafka is designed to handle **high throughput** with **high reliability**. The existence of potentially millions of applications should therefore not cause a problem with respect to data streaming. However, the online learner can become a bottleneck.

In addition, any data streamed through Kafka is **stored persistently** on disk. While Kafka does not necessarily replace traditional databases (indeed Kafka is often used to feed them), it is generally sufficient as a data store and can be used for long-term storage.

Limitations

There are several ways in which my implementation can be improved.

State store. My online learner pushes a continuous stream of parameters to Kafka; even if they are not updated, a new set of parameters is sent to Kafka multiple times per second. In order to avoid this, a state store or a stateful operation should be used. This is not as easy in Kafka as it is with Spark, however, as it requires an additional database.

Message encodings. My messages (such as training data and policy parameters) are exchanged as pure strings (or in fact byte arrays). Ideally, they should be structured according to a schema, which is more robust with respect to further processing. For example, messages could be transformed into json objects. Alternatively, the Avro data format is often used. Kafka provides serializers and deserializers for both formats.

Mutli-node setup. My setup consists of only a single node. A fault-tolerant, high-throughput Kafka cluster will typically comprise more than one node. In addition, both the application(s) and the online learner run on the same machine. In reality, this would not be the case.

Parallel architecture. As already mentioned, I simulate the existence of multiple apps with a loop, which means that strictly speaking, training data is produced sequentially. However, this does not make a real difference as Kafka is asynchronous by design and nothing about my implemented architecture relies on a request-response cycle. Therefore, multiple applications should also be able to run in parallel.

Contextual Bandits. Multi-armed bandits are probably the simplest from of reinforcement learning. While they certainly have important use cases, contextual bandits are even more sophisticated and promising. Generally, my architecture suits the purpose of contextual bandits, too. The only difference is that the training data would consist of (x, a, r) tuples, where x is a context (or feature) vector, and the online learner would be a machine learning model, such as a linear regression model.

References

Academic

Sutton and Barto (2018): Reinforcement Learning: An Introduction.

Lattimore and Szepesvari (2019): Bandit Algorithms

Li, Langford and Schapire (2012): A Contextual-Bandit Approach to Personalized News Article Recommendation.

Practical

Kai Wähner (2019): Machine Learning and Real-Time Analytics in Apache Kafka Applications. Link