Time Travel based Feature Generation

Kedar Sadekar
Netflix Inc.
Los Gatos, CA

Hua Jiang
Netflix Inc.
Los Gatos, CA

ABSTRACT

At Netflix, we are continually looking to improve our member recommendations by following a data driven approach based on machine learning algorithms. To enable faster iterations with high confidence, there are two key components. Collecting historical fact data and providing tools to do feature generation and model training easily.

We describe the components needed for building a time machine for feature generation using Apache Spark that enables our researchers to easily try ideas using historical data and makes running offline experiments and transitioning to A/B tests seamless.

1 INTRODUCTION

We want to make it easy for Netflix members to find great content to fulfill their unique tastes. We follow a two-step approach: first, we try an idea offline using historical data to see if it would have made better recommendations. If it does, we then deploy a live A/B test to see if it performs well in reality, which we measure through statistically significant improvements in core metrics such as member engagement, satisfaction, and retention.

While there are many ways to improve machine learning approaches, arguably the most critical is to provide better input data. A model can only be as good as the data we give it. Most machine learning models expect input to be represented as a vector of numbers, known as a feature vector. Somehow we need to take an arbitrary input entity (e.g. a tuple of member profile, video, country, time, device, etc.), with its associated, richly structured data, and provide a feature vector representing that entity for a machine learning algorithm to use. We call this transformation feature generation and it is central to providing the data needed for learning.

In Section 2, we describe how to store historical data in an accurate yet cost-effective way. In Section 3, we describe how to generate features based on the stored historical data.

2 HISTORICAL FACT DATA

Historical fact data for our members allows us to go back in time and generate features and models for the newer hypotheses'. (Examples of facts are a member's *my list* or *viewing history*). There are various methods by which we could try to collect data for generating features.

(1) *Time versioned store*: Each fact service [1] could maintain a time versioned data store. But this has huge cost implications and the service will not be optimized for serving end user facing requests with low latency.
(2) *Feature logging* During online scoring, we could potentially log all the computed features.
(3) *Pull based model* A pull based model in which we poll the various fact services daily to store the facts.
(4) *Push based model* A push based model where services log temporally accurate facts at time of scoring the member.

The pros and cons of the above approaches are described in [1].

2.1 Snapshots Version 1: Pull-Based Model

Our first approach was the pull based model where we poll the fact services once a for the facts. This has helped us greatly in reducing the time for experimentation, enabling us to turn back the time for the data we capture. Even as it is used successfully in production, this approach presented us with the following challenges:

(1) *Temporal Accuracy* The data may not temporally accurate. The polling happens once a day, whereas the computation of recommendations for our members can happen at different times during the day. Hence, the data used for online scoring and offline model training may not be the same.
(2) *Scale members for facts* To increase the number of members for which facts are stored, the micro-services polled also need to be scaled up to handle the additional load.

2.2 Snapshots Version 2: Push-Based Model

This is our current approach in which services that compute recommendations for our members push facts using a simple logger.log() API allowing us to capture temporally accurate data and meets stratification needs per algorithm.

2.2.1 Challenges and Solutions. With any solution that we develop, we keep in mind cost efficiency (storage, compute) and strive to achieve the right balance / trade-offs between the two.

(1) Volume: With a push based model, 'n' services are logging large number of members and we started seeing huge pressure on our data pipeline [2].
This is being solved by de-duplicating of data being sent (from service) by using a key value store, which also minimizes our final storage.
(2) Facts Usage: Facts stored gets used in two forms. Feature generation pipelines use facts as Java objects (VM based opaque object) whereas for exploratory analysis using Apache Spark, nested structures are preferred. We need to provide API's that can support both use cases.
Our approach is to store in columnar format (Parquet), and convert to POJO's when required. For debugging use cases (random seek) with low latency needs, we alternatively store opaque objects with a TTL in a key value store.

[1] Netflix embraces fine grained SOA, micro-service for each fact

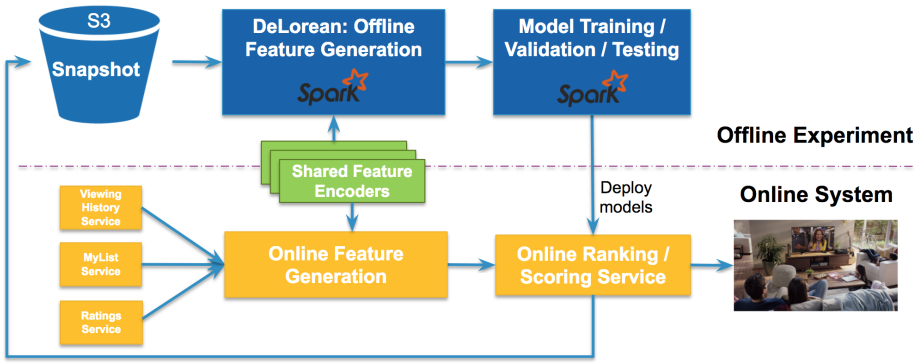

Figure 1: Feature generation for online serving and offline training

3 DELOREAN: OFFLINE FEATURE GENERATOR

3.1 Offline Feature Generation

One of the primary inputs to DeLorean, our offline feature generator, is the label data, which contains information about the contexts (who viewed, in which place, at what time, etc.), items (videos or video groups), and associated labels (watched or not), for which to generate features. Labels are typically the targets used in supervised learning for each context-item combination. For unsupervised learning approaches, the label is not required. As an example, for personalized recommendation the context could be defined as the member profile ID, country code, and time, whereas the recommendation as the video, and the labels as plays or non-plays. In this example, the label data is created by joining the set of snapshotted contexts to the logged play actions.

Data elements are the ingredients that get transformed into features by a feature encoder. Some of these are context-dependent, such as viewing history for a profile, and others are shared by all contexts, such as metadata of the videos. We handle these two types of data elements differently.

For context-dependent data elements, we use the snapshots described above, and associate each one with a data key. We bring all the required snapshot data sources together with the values, items, and labels for each context. Different contexts are broken up to enable distributed feature generation. The snapshots are loaded in a lazy fashion and a series of joins between the label data and all the necessary context-dependent data elements are performed using Apache Spark.

For context-independent data elements, DeLorean broadcasts these bulk data elements to each executor. Since these data elements have manageable sizes and often have a slow rate of change over time, we keep a record of each update that we use to rewind back to the appropriate previous version.

With all required data put together, features can be generated. Once this is done, the data are represented as a Spark DataFrame with an embedded schema. For many personalization application, we need to rank a number of items for each context. To avoid shuffling in the ranking process, item features are grouped by context in the output. The final features are stored in Hive using a Parquet format.

3.2 Separation of Data Retrieval and Data Processing for Easy Deployment to Production

One of the primary motivations for building DeLorean is to share the same feature encoders between offline experiments and online scoring systems to ensure that there are no discrepancies between the features generated for training and those computed online in production. When an idea is ready to be tested online, the model is packaged with the same feature configuration that was used by DeLorean to generate the features.

To compute features in the production system with the model trained offline, we simply switch the data source from online snapshots to online micro-services. We directly call our online micro-services to collect the data elements required by all the feature encoders used in a model and then assemble them into data maps and pass them to the feature encoders. The feature vector is then passed to the offline-trained model for computing predictions, which are used to create our recommendations. Figure 1 shows the high-level process of transitioning from an offline experiment to an online production system where the blocks highlighted in yellow are online systems, and the ones highlighted in blue are offline systems. Note that the feature encoders are shared between online and offline to guarantee the consistency of feature generation.

4 CONCLUSIONS

Fast experimentation is the hallmark of a culture of innovation. Reducing the time to production for an idea is a key metric we use to measure the success of our infrastructure projects. By collecting the state of the online world at a point in time for a select set of contexts, we were able to build a mechanism for turning back time and implement feature generation, model training and validation at scale. DeLorean is now being used in production for feature generation in some of the latest A/B tests for our recommender system.

REFERENCES

[1] Hossein Taghavi, Prasanna Padmanabhan, DB Tsai, Faisal Zakaria Siddiqi, and Justin Basilico. Distributed Time Travel for Feature Generation. https://medium.com/netflix-techblog/distributed-time-travel-for-feature-generation-389cccdd3907

[2] Allen Wang, Steven Wu, Monal Daxini, Manas Alekar, Zhenzhong Xu, Jigish Patel, Nagarjun Guraja, Jonathan Bond, Matt Zimmer, Peter Bakas, and Kunal Kundaje. Kafka Inside Keystone Pipeline. https://medium.com/netflix-techblog/kafka-inside-keystone-pipeline-dd5aeabaf6bb