ROTATIONAL

Think beyond the batch

Building a real-time machine learning application using River and PyEnsign

Agenda

- Overview of batch machine learning and real-time machine learning
- Real-time machine learning use cases
- Advantages of real-time machine learning
- Advantages of streaming databases
- Streaming sentiment analysis application architecture
- Overview of important components of the streaming sentiment analysis application
- Code walkthrough

Batch Machine Learning

- Requires a batch of data to train a model. Often more data implies better model performance.
- Once a model is trained on the entire batch of data, its parameters don't change. It is "**frozen**" to the data at that point of time.
- Follows the **fit→predict** paradigm.

Real-time Machine Learning

- Learns from **one record at a time**.
- Model **learning is dynamic**, parameters adjust every time a model learns on new data.
  - The training cycle does not end
  - The model is continually learning as new data arrives
- Follows the **predict→fit** paradigm.

What types of use cases are real-time models best suited for?

- **Recommender systems** (Netflix movie recommendations, Spotify playlists)
- **Anomaly detection** (Credit card fraud, detecting abnormalities in IoT sensor readings)
- **Ad targeting** (YouTube ads)
- **Content personalization** (Twitter, instagram feeds)
- **Weather forecasting**

Looking for a list of industry specific use cases? **Check out our list** in the docs.

Advantages of real-time machine learning

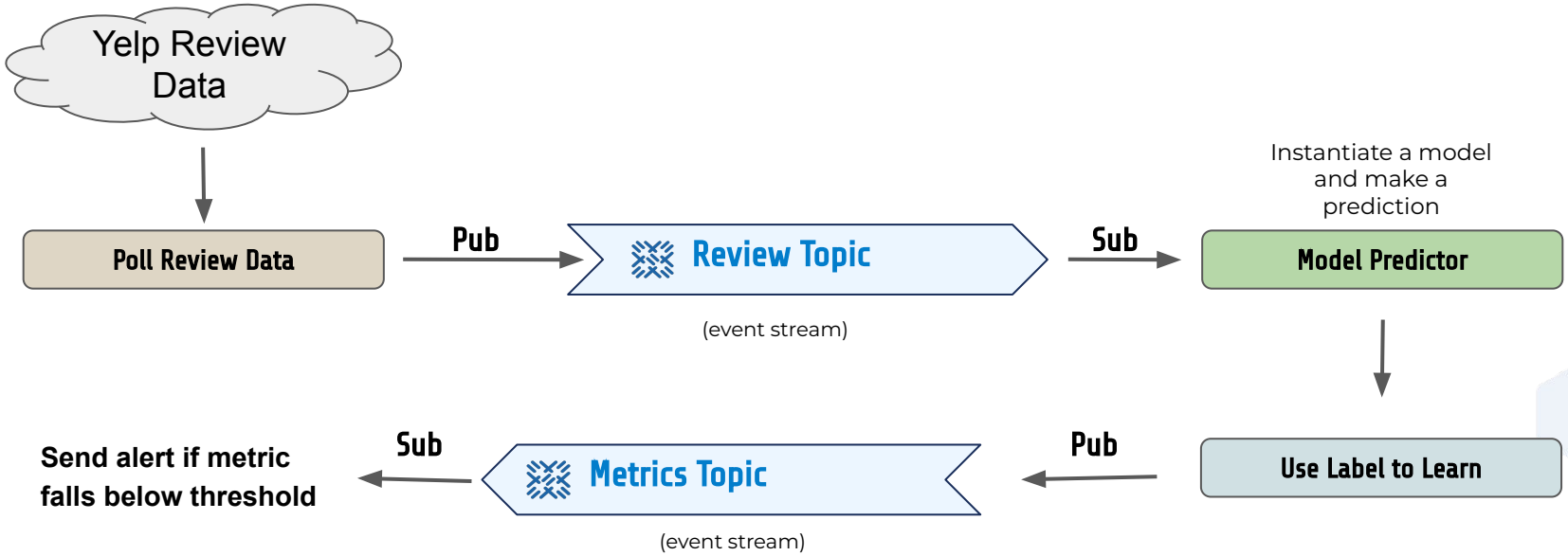
- There is **no need to accumulate a large set of data** to train a model
- Real-time models have a **smaller memory footprint** since they learn incrementally
- Model training and inferences occur on the **same set of features**, which eliminates errors caused by unexpected differences between the training and prediction pipelines
- Real-time models can **train on data as soon as it is available**
  - Eliminates the need for re-training cycles
  - Adapts quickly to changes in the data landscape

Advantages of streaming databases

- Access to **fresh data**
- **Reduce load** on databases used for critical business functions
- Data **cannot get accidentally deleted**
- **Only data needed** for the machine learning application can be published to a topic
  - Publish subset of columns from databases or APIs
  - Mask data prior to publishing
  - No need to do ML development on fake data
- **Decouple data science** from the rest of the application
  - Data scientists can focus on writing machine learning code and not worry about DevOps and application concerns outside of machine learning
  - Simply subscribe to a source data topic and publish machine learning results to a ML results topic

Defining the Data Flows

Streaming Sentiment Analysis Application Architecture



Streaming sentiment analysis application overview

1. How to **initialize** the Ensign client
2. Understanding the asynchronous components of the **publish** and **subscribe functions**

1. Initialize the Ensign Client

Best practice is to not pass in credentials directly

Save the following as environment variables in your OS

- ENSIGN_CLIENT_ID
- ENSIGN_CLIENT_SECRET

```
export ENSIGN_CLIENT_ID="your client id here"
export ENSIGN_CLIENT_SECRET="your client secret here"
```

```
from pyensign.ensign import Ensign
self.ensign = Ensign()
```

The command above will read the values from the OS and initialize the ensign client.

2a. Understanding the Publish function

```
async def publish(self):
```

Note the **async** keyword - this is needed for the program to run asynchronously.

```
await self.ensign.ensure_topic_exists(self.topic)
```

When calling an asynchronous function that does not return immediately, use the **await** keyword.

Creating an **event**:
This example converts a dictionary object to JSON and encodes it as a string (**utf-8**). Specify the mime type to **application/json** since we are publishing a JSON object to the event stream

```
event = Event(json.dumps(record).encode("utf-8"), mimetype="application/json")
```

publish is an asynchronous function so use the **await** keyword.
Specify the topic name, pass the event.
on_ack and **on_nack** are functions you create that specify how to handle acks and nacks from the Ensign server.

```
await self.ensign.publish(self.topic, event, on_ack=handle_ack, on_nack=handle_nack)
```

2b. Understanding the Subscribe function

```
async def subscribe(self):
```

Same as before, the **async** keyword is needed for the program to run asynchronously.

```
await self.ensign.ensure_topic_exists(self.topic)
```

When calling an asynchronous function that does not return immediately, use the **await** keyword.

async for is used to iterate over an **asynchronous generator** which is the asynchronous version of a regular generator. In this example, the subscriber will receive events on the topic as soon as they are available. Once the subscriber receives the event, it passes the event to the **check_metrics** function, which is asynchronous, therefore the call to the function is preceded by **await**.

```
async for event in self.ensign.subscribe(self.topic):
    await self.check_metrics(event)
```

Code Walkthrough