Implementing an Amazon Personalize movie recommender

We built a simple recommendation system using Amazon Personalize by completing the following steps:

1- Data preparation

1.1 Dataset Used:

A sample dataset of user-item interactions was used in CSV format. The raw data was originally from the u.data file (MovieLens 100k) from https://www.kaggle.com/datasets/prajitdatta/movielens-100k-dataset.

1.2 Data Preprocessing:

The dataset required transformation to match Amazon Personalize's expected schema. We made small steps of preprocessing. Here are the following steps we have applied:

1. We loaded the raw tab-separated u.data file using pandas and read it as a df.
2. We selected only the necessary columns: user_id, item_id, and timestamp for the Shema.
3. After that, we converted both user_id and item_id to strings, which is mandatory for Amazon Personalize.
4. And then, we added a constant event_type column with the value "Watch" for all rows, as Amazon Personalize requires an event type.
5. We renamed the columns to match Amazon Personalize's required naming conventions: USER_ID, ITEM_ID, EVENT_TYPE, and TIMESTAMP.
6. And finally, we reordered the columns and used CSV quoting to ensure data type integrity.
7. The final dataset was saved as personalize_interactions_fixed.csv and uploaded to an S3 bucket named my-movie-recommender-data.

1.3 Challenges Faced:

1. **S3 Access Privileges:** There are several dataset import jobs failed due to insufficient S3 access. We initially assumed the bucket policy was correct, but it required additional conditions to match aws: SourceAccount and aws: SourceArn.
2. **IAM Role Debugging:** We used aws iam simulate-principal-policy to verify access permissions and confirmed gaps in the role's policy.
3. **Manual vs GUI Flow:** Frequent switching between the AWS Console and CLI was required to debug access issues, view logs, and monitor status updates.

4. **Trial-and-error Fixes:** We created and deleted multiple import jobs while debugging the permissions problem until the dataset was successfully imported using a widened role policy.

2- IAM role, Policies, Bucket Policies, and schema

IAM User and Permissions

1. We created a user named **adham** because using the root account was restricted for creating IAM roles and granting certain permissions required by Amazon Personalize.
2. **Rationale we agreed on:** IAM best practices recommend using a user instead of root for security and logging.
3. **Credentials that we used:** Access key ID and secret key for the user adham were used to run AWS CLI commands securely.
4. We assigned the following managed policies **directly to the user**:

    1. AdministratorAccess — to ensure full privileges during the test setup.
    2. AmazonPersonalizeFullAccess — it is concrete to working with Amazon Personalize resources.
    3. AmazonS3ReadOnlyAccess — allowed listing and reading the dataset in S3.

IAM Role

Role Name: AmazonPersonalize-ExecutionRole

Trust Policy (linked to personalize.amazonaws.com):

```
{"Version": "2012-10-17",
 "Statement": [
  { "Effect": "Allow",
   "Principal": {
    "Service": "personalize.amazonaws.com" },
   "Action": "sts:AssumeRole"}]}
```

Policies attached to the Role:

1. AmazonPersonalizeFullAccess
2. AmazonS3ReadOnlyAccess
3. final_policy_v2 (a custom inline policy we made for extra flexibility during S3 integration)

our custom policies and their purposes

1- **IAMpolicy** — allowed modifying the bucket policy to grant Amazon Personalize read access:

```
{  "Version": "2012-10-17",
  "Statement": [   {
    "Sid": "VisualEditor0",
    "Effect": "Allow",
    "Action": [
      "s3:PutBucketPolicy",
      "s3:GetBucketPolicy"  ],
    "Resource": "arn:aws:s3:::my-movie-recommender-data"}]}
```

2- **s3_v2** — enabled bucket listing and location access:

```
{ "Version": "2012-10-17",
  "Statement": [
   {"Sid": "VisualEditor0",
    "Effect": "Allow",
    "Action": "s3:GetBucketLocation",
    "Resource": "arn:aws:s3:::my-movie-recommender-data"
   },{
    "Sid": "VisualEditor1",
    "Effect": "Allow",
    "Action": "s3:ListAllMyBuckets","Resource": "*"}]}
```

3- **sts** — granted permission to assume the Personalize execution role:

```
{"Version": "2012-10-17",
 "Statement": [{   "Effect": "Allow",
    "Action": "sts:AssumeRole",
    "Resource": "arn:aws:iam::533267293064:role/AmazonPersonalize-ExecutionRole"}]}
```

4- S3 Bucket Policy — enabled access to dataset by Personalize:

```
{"Version": "2012-10-17", "Statement": [
  {  "Effect": "Allow",
    "Principal": {
      "Service": "personalize.amazonaws.com"
    }, "Action": [
      "s3:GetObject",
      "s3:ListBucket"
    ],"Resource": [
      "arn:aws:s3:::my-movie-recommender-data",
      "arn:aws:s3:::my-movie-recommender-data/*"]}]}
```

5- Amazon Personalize Schema

- **Schema Name:** item_interactions_schema
- **Definition:**

```
{"type": "record",
 "name": "Interactions",
 "namespace": "com.amazonaws.personalize.schema",
 "fields": [
   { "name": "USER_ID", "type": "string" },
   { "name": "ITEM_ID", "type": "string" },
   { "name": "EVENT_TYPE", "type": "string" },
   { "name": "TIMESTAMP", "type": "long" }
 ],"version": "1.0"}
```

3- Model training and recommendations

1. **Create Dataset Group:**

    1. Name: movie_recommendation

2. **Define Schema:**

    1. Name: item_interactions_schema
    2. Defined schema to include USER_ID, ITEM_ID, EVENT_TYPE, TIMESTAMP. I mentioned it earlier with json

3. **Create Dataset:**

    1. Type: INTERACTIONS
    2. Linked with the schema and dataset group.

4. **Import Data:**

    1. Import job: item_interactions_import_job_wide_access
    2. Used the correct IAM role with widened S3 access.

5. **Train Model:**

    1. Solution: movie-recommender-solution
    2. Used-perform-auto-ml to let Personalize choose the best recipe.
    3. AutoML selected the AWS-HRNN recipe.
    4. Solution version took ~20 minutes to train.

6. **Create Campaign:**

    1. Campaign: movie-recommender-campaign
    2. Provisioned TPS: 1
    3. Status changed to ACTIVE after ~45 minutes.

7. **Generate Recommendations:**

    1. Used CLI to invoke GetRecommendations for sample user ID 22.
    2. Returned a ranked list of recommended item IDs with scores.

Challenges Faced:

1. **AutoML and Recipe Conflict:** We mistakenly specified a recipe along with --perform-auto-ml, which caused a failure. Only one approach can be used.

2. **Delayed Campaign Activation:** The campaign creation took much longer than expected (~45 minutes), leading us to attempt troubleshooting with CloudWatch.
3. **CloudWatch Log Setup:** We tried enabling CloudWatch logging via CloudTrail but faced issues with ARN validation. Ultimately, we relied on CLI and console for debugging.
4. **CLI Commands Mastery:** Several CLI prompts had syntax or naming errors (like --min-provisionedTPS instead of --min-provisioned-tps) that required careful reading of documentation and trial corrections.

4- Recommendations generation

The model successfully returned relevant recommendations, and the differences in confidence scores highlight the diversity of user behavior captured by the recommender.

1. **User 22:** Top recommendation was item 250 with the highest score of 0.0512. The rest included 541, 797, 586, showing a preference for possibly popular or highly rated movies.
2. **User 6:** Received item 315 as top recommendation with a significantly higher score 0.0897, indicating a more confident prediction. Items 310, 272, and 751 followed closely.
3. **User 122:** Showed relatively lower prediction scores with top recommendation being item 88 with a score of 0.0131. This may suggest sparse historical interaction or less distinct preference patterns.

5- Short deployment plan

We need to integrate Amazon Personalize recommendations into a real-world application. We should propose the following deployment plan:

1. Backend Setup via AWS Lambda and API Gateway

    1. Create a REST API in Amazon API Gateway.
    2. Connect it to an AWS Lambda function written in Python or Node.js.
    3. The Lambda function calls the Amazon Personalize campaign using the GetRecommendations API.

2. Frontend Integration

    1. The frontend will send a request to the API with the user's ID to begain.

2. The backend will return the personalized items accurately, which should be displayed as recommended content.

3. Security and Scaling

    1. We use IAM roles with minimal permissions for Lambda.
    2. And then we enable caching or pre-fetching recommendations for high-traffic users.

4. Cost Management

    1. We schedule the cleanup of unused campaigns and solutions.
    2. After that, we monitor usage via AWS Billing, Budgets, and Cost Explorer.