AI Use cases

Thursday, December 12, 2024

11:11 AM

### Use Case: Personalized News Recommendation System

\*\*Objective:\*\*

Build a personalized news recommendation system that:

1. Collects user interaction data (e.g., reads, likes, shares) on news articles.

2. Uses machine learning to recommend news articles based on user preferences.

3. Provides text-to-speech functionality for reading articles aloud.

4. Monitors application performance and user interactions.

### High-Level Architecture:

1. \*\*Data Collection\*\*: Use Amazon Kinesis Streams to capture user interaction events.

2. \*\*Data Storage and Preprocessing\*\*: Store data in Amazon S3 for preprocessing.

3. \*\*Machine Learning\*\*: Use Amazon SageMaker to train and deploy a recommendation model.

4. \*\*Recommendation and Serving\*\*: Use Amazon Personalize to generate recommendations.

5. \*\*Text-to-Speech\*\*: Use Amazon Polly for converting text articles to speech.

6. \*\*Monitoring and Logging\*\*: Use Amazon CloudWatch for monitoring and logging.

### Project Components and Workflow:

#### 1. Data Collection

- \*\*Amazon Kinesis Streams\*\*: Capture user interactions with news articles.

```java

import com.amazonaws.services.kinesis.AmazonKinesis;

import com.amazonaws.services.kinesis.AmazonKinesisClientBuilder;

import com.amazonaws.services.kinesis.model.PutRecordRequest;

public class KinesisDataStreamer {

private static final String STREAM\_NAME = "user-interaction-stream";

private static final AmazonKinesis kinesisClient = AmazonKinesisClientBuilder.defaultClient();

public void publishInteractionEvent(String userId, String newsId, String action) {

String data = String.format("{\"userId\":\"%s\", \"newsId\":\"%s\", \"action\":\"%s\"}", userId, newsId, action);

PutRecordRequest request = new PutRecordRequest()

.withStreamName(STREAM\_NAME)

.withPartitionKey(userId)

.withData(ByteBuffer.wrap(data.getBytes()));

kinesisClient.putRecord(request);

}

}

```

#### 2. Data Storage and Preprocessing

- \*\*Amazon S3\*\*: Store raw interaction data for preprocessing.

```java

import com.amazonaws.services.s3.AmazonS3;

import com.amazonaws.services.s3.AmazonS3ClientBuilder;

import com.amazonaws.services.s3.model.PutObjectRequest;

public class S3DataUploader {

private static final String BUCKET\_NAME = "user-interactions-bucket";

private static final AmazonS3 s3Client = AmazonS3ClientBuilder.defaultClient();

public void uploadData(String key, File dataFile) {

s3Client.putObject(new PutObjectRequest(BUCKET\_NAME, key, dataFile));

}

}

```

#### 3. Machine Learning

- \*\*Amazon SageMaker\*\*: Train a recommendation model using the historical interaction data.

```python

# Jupyter Notebook on SageMaker instance

import sagemaker

from sagemaker import get\_execution\_role

role = get\_execution\_role()

session = sagemaker.Session()

# Create and configure estimator

from sagemaker.estimator import Estimator

estimator = Estimator(image\_uri='your-custom-image-uri',

role=role,

instance\_count=1,

instance\_type='ml.m5.large',

output\_path='s3://path-to-output')

# Launch training

estimator.fit({'train': 's3://path-to-training-data'})

```

#### 4. Recommendation and Serving

- \*\*Amazon Personalize\*\*: Generate personalized recommendations.

```java

import com.amazonaws.services.personalize.AmazonPersonalize;

import com.amazonaws.services.personalize.AmazonPersonalizeClientBuilder;

import com.amazonaws.services.personalize.model.GetRecommendationsRequest;

import com.amazonaws.services.personalize.model.GetRecommendationsResult;

public class PersonalizeService {

private static final String CAMPAIGN\_ARN = "your-campaign-arn";

private AmazonPersonalize personalizeClient = AmazonPersonalizeClientBuilder.defaultClient();

public List<String> getRecommendations(String userId) {

GetRecommendationsRequest request = new GetRecommendationsRequest()

.withCampaignArn(CAMPAIGN\_ARN)

.withUserId(userId);

GetRecommendationsResult result = personalizeClient.getRecommendations(request);

return result.getItemList().stream()

.map(item -> item.getItemId())

.collect(Collectors.toList());

}

}

```

#### 5. Text-to-Speech

- \*\*Amazon Polly\*\*: Convert recommended news articles to speech.

```java

import com.amazonaws.services.polly.AmazonPolly;

import com.amazonaws.services.polly.AmazonPollyClientBuilder;

import com.amazonaws.services.polly.model.OutputFormat;

import com.amazonaws.services.polly.model.SynthesizeSpeechRequest;

import com.amazonaws.services.polly.model.SynthesizeSpeechResult;

import java.io.InputStream;

public class PollyService {

private final AmazonPolly pollyClient = AmazonPollyClientBuilder.defaultClient();

public InputStream synthesizeSpeech(String text) {

SynthesizeSpeechRequest request = new SynthesizeSpeechRequest()

.withText(text)

.withOutputFormat(OutputFormat.Mp3)

.withVoiceId("Joanna");

SynthesizeSpeechResult result = pollyClient.synthesizeSpeech(request);

return result.getAudioStream();

}

}

```

#### 6. Monitoring and Logging

- \*\*Amazon CloudWatch\*\*: Monitor application performance and log user interactions.

```java

import com.amazonaws.services.cloudwatch.AmazonCloudWatch;

import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;

import com.amazonaws.services.cloudwatch.model.PutMetricDataRequest;

import com.amazonaws.services.cloudwatch.model.MetricDatum;

import com.amazonaws.services.cloudwatch.model.StandardUnit;

public class CloudWatchMetrics {

private final AmazonCloudWatch cloudWatchClient = AmazonCloudWatchClientBuilder.defaultClient();

public void publishMetric(String metricName, double value) {

PutMetricDataRequest request = new PutMetricDataRequest()

.withNamespace("NewsRecommendationSystem")

.withMetricData(new MetricDatum()

.withMetricName(metricName)

.withUnit(StandardUnit.Count)

.withValue(value));

cloudWatchClient.putMetricData(request);

}

}

```

### Putting It All Together

Main.java:

```java

public class Main {

public static void main(String[] args) {

// Data collection

KinesisDataStreamer dataStreamer = new KinesisDataStreamer();

dataStreamer.publishInteractionEvent("user123", "news456", "read");

// Recommendation

PersonalizeService personalizeService = new PersonalizeService();

List<String> recommendations = personalizeService.getRecommendations("user123");

System.out.println("Recommended items: " + recommendations);

// Text-to-Speech

PollyService pollyService = new PollyService();

InputStream speechStream = pollyService.synthesizeSpeech("Welcome to your personalized news feed.");

// Play the audio stream using an appropriate media player library

// Monitoring

CloudWatchMetrics cloudWatchMetrics = new CloudWatchMetrics();

cloudWatchMetrics.publishMetric("UserReadEvents", 1);

// Data upload (for offline processing like training)

S3DataUploader s3Uploader = new S3DataUploader();

s3Uploader.uploadData("interactions/data.json", new File("path/to/data.json"));

}

}

```

### Conclusion

This example demonstrates a comprehensive Java-based application integrating AWS AI tools and services. You can further extend this project by incorporating more sophisticated machine learning models, adding error handling, securing data transmission, and creating rich user interfaces. This setup provides a robust framework to build upon for creating powerful AI-driven applications on AWS.

Certainly! Let’s expand the use case with additional scenarios and provide more in-depth code examples, especially focusing on real-time data handling. Here’s an extended and detailed look at a personalized news recommendation system.

### Additional Scenarios:

1. \*\*Real-time Data Capture\*\*: Capture and buffer user interaction data using Amazon Kinesis Data Streams.

2. \*\*Real-time Data Processing\*\*: Process streamed data using AWS Lambda.

3. \*\*Model Training and Deployment\*\*: Train a recommendation model with Amazon SageMaker.

4. \*\*Generating Real-time Recommendations\*\*: Serve recommendations using Amazon Personalize.

5. \*\*Text-to-Speech Conversion\*\*: Convert articles to speech using Amazon Polly.

6. \*\*Monitoring and Logging\*\*: Monitor system performance and log interactions using Amazon CloudWatch.

### Detailed Implementation:

#### 1. Real-time Data Capture Using Kinesis Data Streams

\*\*Setup your Kinesis Data Stream\*\*: You can create a Kinesis Stream using AWS Management Console, AWS CLI, or SDKs.

\*\*Java Code to Publish Data to Kinesis Stream\*\*

```java

import com.amazonaws.auth.AWSStaticCredentialsProvider;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;

import com.amazonaws.regions.Regions;

import com.amazonaws.services.kinesis.AmazonKinesis;

import com.amazonaws.services.kinesis.AmazonKinesisClientBuilder;

import com.amazonaws.services.kinesis.model.PutRecordRequest;

import com.amazonaws.services.kinesis.model.PutRecordResult;

import java.nio.ByteBuffer;

public class KinesisDataStreamer {

private static final String STREAM\_NAME = "user-interaction-stream";

private static final Regions REGION = Regions.US\_EAST\_1;

private AmazonKinesis kinesisClient;

public KinesisDataStreamer() {

this.kinesisClient = AmazonKinesisClientBuilder.standard()

.withRegion(REGION)

.withCredentials(new ProfileCredentialsProvider())

.build();

}

public void publishInteractionEvent(String userId, String newsId, String action) {

String data = String.format("{\"userId\":\"%s\", \"newsId\":\"%s\", \"action\":\"%s\"}", userId, newsId, action);

ByteBuffer dataBuffer = ByteBuffer.wrap(data.getBytes());

PutRecordRequest putRecordRequest = new PutRecordRequest()

.withStreamName(STREAM\_NAME)

.withPartitionKey(userId)

.withData(dataBuffer);

try {

PutRecordResult putRecordResult = kinesisClient.putRecord(putRecordRequest);

System.out.println("Successfully put record into stream: " + putRecordResult.getSequenceNumber());

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

KinesisDataStreamer dataStreamer = new KinesisDataStreamer();

dataStreamer.publishInteractionEvent("user123", "news456", "read");

}

}

```

#### 2. Real-time Data Processing Using AWS Lambda

Create an AWS Lambda function to process data from the Kinesis stream. You can use the AWS Lambda console to create the function and configure it to trigger on new records in the Kinesis stream.

\*\*Lambda Function Code (Java)\*\*:

```java

import com.amazonaws.services.lambda.runtime.Context;

import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

import com.amazonaws.services.lambda.runtime.RequestHandler;

import com.amazonaws.services.lambda.runtime.events.KinesisEvent.KinesisEventRecord;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;

import com.amazonaws.services.dynamodbv2.document.DynamoDB;

import com.amazonaws.services.dynamodbv2.document.Table;

import com.amazonaws.services.dynamodbv2.document.PutItemOutcome;

import com.amazonaws.services.dynamodbv2.document.Item;

import java.nio.charset.StandardCharsets;

public class KinesisLambdaHandler implements RequestHandler<KinesisEvent, String> {

private final AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

private final DynamoDB dynamoDB = new DynamoDB(client);

private static final String TABLE\_NAME = "user-interactions";

@Override

public String handleRequest(KinesisEvent event, Context context) {

Table table = dynamoDB.getTable(TABLE\_NAME);

for (KinesisEventRecord record : event.getRecords()) {

String data = new String(record.getKinesis().getData().array(), StandardCharsets.UTF\_8);

// Assuming the Kinesis record data is JSON

Item item = Item.fromJSON(data);

PutItemOutcome outcome = table.putItem(item);

context.getLogger().log("Processed record into table: " + outcome.getPutItemResult().toString());

}

return "Processed " + event.getRecords().size() + " records.";

}

}

```

#### 3. Model Training and Deployment Using Amazon SageMaker

You can use a Jupyter Notebook in Amazon SageMaker for training and deploying the model. Below is a Python code snippet to be used in the notebook.

\*\*Python Code Snippet for SageMaker Notebook\*\*:

```python

import sagemaker

from sagemaker import get\_execution\_role

from sagemaker.estimator import Estimator

role = get\_execution\_role()

session = sagemaker.Session()

bucket = 'your-s3-bucket-name'

prefix = 'sagemaker/news-recommendation'

# Upload the training data to S3

training\_data\_uri = session.upload\_data(path='data/train\_data.csv', bucket=bucket, key\_prefix=prefix + '/train')

# Define the model estimator

xgboost\_container = sagemaker.image\_uris.retrieve('xgboost', session.boto\_region\_name, '1.2-1')

xgb = Estimator(image\_uri=xgboost\_container,

role=role,

instance\_count=1,

instance\_type='ml.m5.large',

output\_path='s3://{}/{}/output'.format(bucket, prefix),

sagemaker\_session=session)

# Set hyperparameters

xgb.set\_hyperparameters(objective='reg:logistic',

num\_round=100)

# Train the model

xgb.fit({'train': training\_data\_uri})

# Deploy the model as a SageMaker endpoint

xgb\_predictor = xgb.deploy(initial\_instance\_count=1, instance\_type='ml.m5.large')

```

#### 4. Generating Real-time Recommendations Using Amazon Personalize

Set up Amazon Personalize to train and deploy models for real-time recommendations.

\*\*Java Code to Get Recommendations from Amazon Personalize\*\*:

```java

import com.amazonaws.services.personalize.AmazonPersonalize;

import com.amazonaws.services.personalize.AmazonPersonalizeClientBuilder;

import com.amazonaws.services.personalize.model.GetRecommendationsRequest;

import com.amazonaws.services.personalize.model.GetRecommendationsResult;

import com.amazonaws.services.personalize.model.PredictionItem;

import java.util.List;

import java.util.stream.Collectors;

public class PersonalizeService {

private static final String CAMPAIGN\_ARN = "your-campaign-arn";

private AmazonPersonalize personalizeClient;

public PersonalizeService() {

this.personalizeClient = AmazonPersonalizeClientBuilder.standard().build();

}

public List<String> getRecommendations(String userId) {

GetRecommendationsRequest request = new GetRecommendationsRequest()

.withCampaignArn(CAMPAIGN\_ARN)

.withUserId(userId);

GetRecommendationsResult result = personalizeClient.getRecommendations(request);

return result.getItemList().stream()

.map(PredictionItem::getItemId)

.collect(Collectors.toList());

}

public static void main(String[] args) {

PersonalizeService personalizeService = new PersonalizeService();

List<String> recommendations = personalizeService.getRecommendations("user123");

recommendations.forEach(System.out::println);

}

}

```

#### 5. Text-to-Speech Conversion Using Amazon Polly

\*\*Java Code to Convert Text to Speech Using Amazon Polly\*\*:

```java

import com.amazonaws.services.polly.AmazonPolly;

import com.amazonaws.services.polly.AmazonPollyClientBuilder;

import com.amazonaws.services.polly.model.OutputFormat;

import com.amazonaws.services.polly.model.SynthesizeSpeechRequest;

import com.amazonaws.services.polly.model.SynthesizeSpeechResult;

import java.io.InputStream;

public class PollyService {

private final AmazonPolly pollyClient;

public PollyService() {

this.pollyClient = AmazonPollyClientBuilder.standard().withRegion("us-east-1").build();

}

public InputStream synthesizeSpeech(String text) {

SynthesizeSpeechRequest synthesizeSpeechRequest = new SynthesizeSpeechRequest()

.withText(text)

.withOutputFormat(OutputFormat.Mp3)

.withVoiceId("Joanna");

SynthesizeSpeechResult synthesizeSpeechResult = pollyClient.synthesizeSpeech(synthesizeSpeechRequest);

return synthesizeSpeechResult.getAudioStream();

}

public static void main(String[] args) {

PollyService pollyService = new PollyService();

try (InputStream speechStream = pollyService.synthesizeSpeech("Welcome to your personalized news feed.")) {

// You can save the audio stream to a file or play it directly.

} catch (Exception e) {

e.printStackTrace();

}

}

}

```

#### 6. Monitoring and Logging Using Amazon CloudWatch

\*\*Java Code to Publish Custom Metrics to CloudWatch\*\*:

```java

import com.amazonaws.services.cloudwatch.AmazonCloudWatch;

import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;

import com.amazonaws.services.cloudwatch.model.Dimension;

import com.amazonaws.services.cloudwatch.model.MetricDatum;

import com.amazonaws.services.cloudwatch.model.PutMetricDataRequest;

import com.amazonaws.services.cloudwatch.model.StandardUnit;

public class CloudWatchMetrics {

private final AmazonCloudWatch cloudWatchClient;

public CloudWatchMetrics() {

this.cloudWatchClient = AmazonCloudWatchClientBuilder.defaultClient();

}

public void publishMetric(String metricName, double value) {

MetricDatum datum = new MetricDatum()

.withMetricName(metricName)

.withUnit(StandardUnit.Count)

.withValue(value)

.withDimensions(new Dimension().withName("AppName").withValue("NewsRecommendationSystem"));

PutMetricDataRequest request = new PutMetricDataRequest()

.withNamespace("NewsRecommendationSystem")

.withMetricData(datum);

cloudWatchClient.putMetricData(request);

}

public static void main(String[] args) {

CloudWatchMetrics cloudWatchMetrics = new CloudWatchMetrics();

cloudWatchMetrics.publishMetric("UserReadEvents", 1);

}

}

```

### Putting It All Together

Integrate the components into a coherent application.

```java

public class Main {

public static void main(String[] args) {

// 1. Data collection - Publish a fake user interaction event

KinesisDataStreamer dataStreamer = new KinesisDataStreamer();

dataStreamer.publishInteractionEvent("user123", "news456", "read");

// 2. Recommendation - Get recommendations for the user

PersonalizeService personalizeService = new PersonalizeService();

List<String> recommendations = personalizeService.getRecommendations("user123");

System.out.println("Recommended news articles: " + recommendations);

// 3. Text-to-Speech - Convert the first recommended article to speech

PollyService pollyService = new PollyService();

if (!recommendations.isEmpty()) {

try (InputStream speechStream = pollyService.synthesizeSpeech("This is a recommended news article.")) {

// Save or play the audio stream

// Example: Save to a file

File targetFile = new File("output.mp3");

java.nio.file.Files.copy(

speechStream,

targetFile.toPath(),

java.nio.file.StandardCopyOption.REPLACE\_EXISTING);

} catch (Exception e) {

e.printStackTrace();

}

}

// 4. Monitoring - Publish a custom metric to CloudWatch

CloudWatchMetrics cloudWatchMetrics = new CloudWatchMetrics();

cloudWatchMetrics.publishMetric("UserReadEvents", 1);

// Additional Integration Code (real-time data integration, processing, etc.)

}

}

```

### Conclusion

This extended example provides a comprehensive, end-to-end Java-based application integrating various AWS AI tools and services. The application captures real-time user interaction data, processes it, generates recommendations, converts text to speech, and monitors the system's performance. This setup demonstrates how AWS services can be used together to build a robust, scalable, AI-driven application.

Certainly! Below is a detailed walkthrough of how to create a Java project utilizing NLP for a retail application that generates top recommended items. This project will use AWS services such as SageMaker, CloudWatch, Textract, Glue, and others to build the solution.

### Project Overview

- \*\*Objective\*\*: Develop a retail search application with NLP capabilities to understand user queries and generate top recommended items.

- \*\*Services Used\*\*:

- Amazon SageMaker: For training and deploying a NLP model for understanding user intents.

- AWS Glue: For ETL (Extract, Transform, Load) processes to prepare data.

- Amazon Textract: For extracting text from scanned documents (optional if adding OCR capabilities).

- Amazon CloudWatch: For monitoring and logging.

- Amazon S3: For storing processed data and other assets.

- Amazon DynamoDB or RDS: For storing product data and user interactions (optional - depends on specific data requirements).

### High-Level Steps

1. \*\*Data Preparation and ETL with AWS Glue\*\*: We will use AWS Glue to process and prepare product and interaction data.

2. \*\*NLP Model Training with Amazon SageMaker\*\*: Train a machine learning model to understand user queries and recommend products.

3. \*\*Application Development\*\*: Implement the Java application to interact with the AWS services.

4. \*\*Monitoring with Amazon CloudWatch\*\*: Set up monitoring and logging to observe application performance.

5. \*\*Optional: Adding OCR capabilities with Textract\*\*: Enhance the application to extract text from documents if required.

### Detailed Implementation

#### 1. Data Preparation and ETL with AWS Glue

First, let's prepare and clean the product data using AWS Glue.

\*\*Glue ETL Script Example (Python script to be used in Glue Job)\*\*:

```python

import sys

from awsglue.transforms import \*

from awsglue.utils import getResolvedOptions

from pyspark.context import SparkContext

from awsglue.context import GlueContext

from awsglue.job import Job

args = getResolvedOptions(sys.argv, ['JOB\_NAME'])

sc = SparkContext()

glueContext = GlueContext(sc)

spark = glueContext.spark\_session

job = Job(glueContext)

job.init(args['JOB\_NAME'], args)

# Load product dataset from S3

datasource0 = glueContext.create\_dynamic\_frame.from\_catalog(

database = "retail\_db", table\_name = "product\_data"

)

# Perform basic cleaning and transformation

applymapping1 = ApplyMapping.apply(

frame = datasource0,

mappings = [

("product\_id", "string", "product\_id", "string"),

("product\_name", "string", "product\_name", "string"),

("category", "string", "category", "string"),

("price", "double", "price", "double"),

("stock", "int", "stock", "int")

]

)

# Write transformed data back to S3

s3\_sink = glueContext.getSink(

path = "s3://processed-product-data/output/",

connection\_type = "s3",

updateBehavior = "UPDATE\_IN\_DATABASE",

partitionKeys = [],

compression = "snappy",

format = "parquet"

)

s3\_sink.setCatalogInfo(catalogDatabase="retail\_db", catalogTableName="processed\_product\_data")

s3\_sink.writeFrame(applymapping1)

job.commit()

```

#### 2. NLP Model Training with Amazon SageMaker

Use a pre-trained NLP model or train your own using SageMaker. For simplicity, let’s use a pre-trained BERT model for intent recognition.

\*\*SageMaker Training Script Example (Python)\*\*:

```python

# Assuming you have a dataset already formatted for training

import sagemaker

from sagemaker import get\_execution\_role

role = get\_execution\_role()

session = sagemaker.Session()

bucket = 'your-s3-bucket-name'

prefix = 'sagemaker/nlp-model'

estimator = sagemaker.estimator.Estimator(

'763104351884.dkr.ecr.us-west-2.amazonaws.com/huggingface-pytorch-training:1.6-transformers4.4-gpu-py36-cu110-ubuntu18.04',

role,

instance\_count=1,

instance\_type='ml.p3.2xlarge',

output\_path='s3://{}/{}/output'.format(bucket, prefix),

sagemaker\_session=session

)

estimator.fit({

'train': 's3://{}/{}/train'.format(bucket, prefix),

'test': 's3://{}/{}/test'.format(bucket, prefix)

})

predictor = estimator.deploy(initial\_instance\_count=1, instance\_type='ml.m5.large')

```

#### 3. Application Development in Java

\*\*Example Java Code\*\*:

\*\*Maven Dependencies (pom.xml)\*\*:

```xml

<dependencies>

<!-- AWS SDK dependencies -->

<dependency>

<groupId>com.amazonaws</groupId>

<artifactId>aws-java-sdk-sagemaker</artifactId>

<version>1.11.1034</version>

</dependency>

<dependency>

<groupId>com.amazonaws</groupId>

<artifactId>aws-java-sdk-cloudwatch</artifactId>

<version>1.11.1034</version>

</dependency>

<dependency>

<groupId>com.amazonaws</groupId>

<artifactId>aws-java-sdk-textract</artifactId>

<version>1.11.1034</version>

</dependency>

<!-- Other dependencies such as logging or JSON handling -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.30</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>1.7.30</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.12.3</version>

</dependency>

</dependencies>

```

\*\*SageMaker Predictor for Sentiment Analysis\*\*:

```java

import com.amazonaws.auth.AWSStaticCredentialsProvider;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;

import com.amazonaws.services.sagemakerruntime.AmazonSageMakerRuntime;

import com.amazonaws.services.sagemakerruntime.AmazonSageMakerRuntimeClientBuilder;

import com.amazonaws.services.sagemakerruntime.model.InvokeEndpointRequest;

import com.amazonaws.services.sagemakerruntime.model.InvokeEndpointResult;

import java.nio.charset.StandardCharsets;

import java.util.Base64;

public class SageMakerPredictor {

private static final String ENDPOINT\_NAME = "your-sagemaker-endpoint";

private final AmazonSageMakerRuntime sageMakerRuntime;

public SageMakerPredictor() {

this.sageMakerRuntime = AmazonSageMakerRuntimeClientBuilder.standard()

.withRegion("us-west-2")

.withCredentials(new ProfileCredentialsProvider())

.build();

}

public String predictSentiment(String text) {

InvokeEndpointRequest invokeEndpointRequest = new InvokeEndpointRequest()

.withEndpointName(ENDPOINT\_NAME)

.withContentType("application/json")

.withBody(Base64.getEncoder().encodeToString(text.getBytes(StandardCharsets.UTF\_8)));

InvokeEndpointResult result = sageMakerRuntime.invokeEndpoint(invokeEndpointRequest);

return new String(Base64.getDecoder().decode(result.getBody().array()), StandardCharsets.UTF\_8);

}

public static void main(String[] args) {

SageMakerPredictor predictor = new SageMakerPredictor();

String text = "I want to buy a new laptop";

String result = predictor.predictSentiment(text);

System.out.println("Predicted Sentiment: " + result);

}

}

```

\*\*Product Recommendations Based on User Queries\*\*:

```java

import java.util.List;

import java.util.ArrayList;

import java.util.Map;

import java.util.HashMap;

public class RecommendationService {

private SageMakerPredictor predictor;

public RecommendationService() {

this.predictor = new SageMakerPredictor();

}

public List<String> getRecommendations(String userQuery) {

// Perform NLP on userQuery to understand intent

String intent = predictor.predictSentiment(userQuery).toLowerCase();

// A simplified product database for demonstration

Map<String, List<String>> productDatabase = new HashMap<>();

productDatabase.put("laptop", List.of("Laptop1", "Laptop2", "Laptop3"));

productDatabase.put("phone", List.of("Phone1", "Phone2", "Phone3"));

productDatabase.put("headphone", List.of("Headphone1", "Headphone2", "Headphone3"));

// Find matching products

return productDatabase.getOrDefault(intent, new ArrayList<>());

}

public static void main(String[] args) {

RecommendationService service = new RecommendationService();

String query = "I am looking for a new smartphone";

List<String> recommendations = service.getRecommendations(query);

recommendations.forEach(System.out::println);

}

}

```

#### 4. Monitoring with Amazon CloudWatch

\*\*Java Code to Publish Custom Metrics to CloudWatch\*\*:

```java

import com.amazonaws.services.cloudwatch.AmazonCloudWatch;

import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;

import com.amazonaws.services.cloudwatch.model.Dimension;

import com.amazonaws.services.cloudwatch.model.MetricDatum;

import com.amazonaws.services.cloudwatch.model.PutMetricDataRequest;

import com.amazonaws.services.cloudwatch.model.StandardUnit;

public class CloudWatchMetrics {

private final AmazonCloudWatch cloudWatchClient;

public CloudWatchMetrics() {

this.cloudWatchClient = AmazonCloudWatchClientBuilder.defaultClient();

}

public void publishMetric(String metricName, double value) {

MetricDatum datum = new MetricDatum()

.withMetricName(metricName)

.withUnit(StandardUnit.Count)

.withValue(value)

.withDimensions(new Dimension().withName("AppName").withValue("RetailSearchApp"));

PutMetricDataRequest request = new PutMetricDataRequest()

.withNamespace("RetailSearchAppMetrics")

.withMetricData(datum);

cloudWatchClient.putMetricData(request);

}

public static void main(String[] args) {

CloudWatchMetrics metrics = new CloudWatchMetrics();

metrics.publishMetric("UserSearchQueries", 1);

}

}

```

### Full Application Setup

Integrate the components into a coherent application. If you need to add OCR capabilities using Amazon Textract, you can extend the example further.

\*\*Main Java Application\*\*:

```java

public class Main {

public static void main(String[] args) {

// Initialize services

RecommendationService recommendationService = new RecommendationService();

CloudWatchMetrics cloudWatchMetrics = new CloudWatchMetrics();

// Simulate a user search query

String userQuery = "I want to buy a new laptop";

List<String> recommendations = recommendationService.getRecommendations(userQuery);

// Output recommendations

System.out.println("Top Recommendations:");

recommendations.forEach(System.out::println);

// Publish a custom metric to CloudWatch

cloudWatchMetrics.publishMetric("UserSearchQueries", 1);

}

}

```

### Conclusion

This example provides a comprehensive, real-world Java-based application integrating AWS AI tools and services to build a retail NLP search and recommendation system. The setup demonstrates how to prepare data, train an NLP model, implement recommendations, and monitor application performance using AWS services such as SageMaker, Glue, Textract, and CloudWatch. You can further extend this project by adding more sophisticated data preprocessing, model training, and error-handling mechanisms.

Sure! Let's create a Java project that integrates AWS Personalize into a retail application to generate product recommendations based on user interactions and search queries. We'll go through setting up the necessary services and implementing the Java code that interacts with Amazon Personalize.

### Overview

We'll follow these steps:

1. \*\*Data Collection and Preparation\*\*: Use AWS Glue to process and prepare interaction data.

2. \*\*Setting Up Amazon Personalize\*\*: Create datasets, solutions, and campaigns in Amazon Personalize.

3. \*\*Java Application Development\*\*: Develop a Java application to interact with Amazon Personalize for generating recommendations.

4. \*\*Monitoring and Logging\*\*: Use CloudWatch to monitor and log application performance.

### Step-by-Step Implementation

#### 1. Data Collection and Preparation with AWS Glue

We'll use AWS Glue to process and prepare the interaction data. Here is an example of a Glue job script in Python:

\*\*Glue ETL Script Example (Python)\*\*:

```python

import sys

from awsglue.transforms import \*

from awsglue.utils import getResolvedOptions

from pyspark.context import SparkContext

from awsglue.context import GlueContext

from awsglue.job import Job

args = getResolvedOptions(sys.argv, ['JOB\_NAME'])

sc = SparkContext()

glueContext = GlueContext(sc)

spark = glueContext.spark\_session

job = Job(glueContext)

job.init(args['JOB\_NAME'], args)

# Load interaction dataset from S3

datasource0 = glueContext.create\_dynamic\_frame.from\_catalog(

database = "retail\_db", table\_name = "user\_interactions"

)

# Perform basic cleaning and transformation

applymapping1 = ApplyMapping.apply(

frame = datasource0,

mappings = [

("user\_id", "string", "user\_id", "string"),

("item\_id", "string", "item\_id", "string"),

("interaction\_type", "string", "interaction\_type", "string"),

("timestamp", "long", "timestamp", "long")

]

)

# Write transformed data back to S3

s3\_sink = glueContext.getSink(

path = "s3://processed-interaction-data/output/",

connection\_type = "s3",

updateBehavior = "UPDATE\_IN\_DATABASE",

partitionKeys = [],

compression = "snappy",

format = "parquet"

)

s3\_sink.setCatalogInfo(catalogDatabase="retail\_db", catalogTableName="processed\_user\_interactions")

s3\_sink.writeFrame(applymapping1)

job.commit()

```

#### 2. Setting Up Amazon Personalize

Next, set up Amazon Personalize by creating datasets, solutions, and campaigns. This can be done using the AWS SDK for Java or manually through the AWS Management Console.

\*\*Java Code to Set Up Amazon Personalize\*\*:

```java

import com.amazonaws.auth.AWSStaticCredentialsProvider;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;

import com.amazonaws.regions.Regions;

import com.amazonaws.services.personalize.AmazonPersonalize;

import com.amazonaws.services.personalize.AmazonPersonalizeClientBuilder;

import com.amazonaws.services.personalize.model.\*;

public class PersonalizeSetup {

private AmazonPersonalize personalizeClient;

public PersonalizeSetup() {

this.personalizeClient = AmazonPersonalizeClientBuilder.standard()

.withRegion(Regions.US\_WEST\_2)

.withCredentials(new ProfileCredentialsProvider())

.build();

}

public void createPersonalizeResources() {

// Create dataset group

CreateDatasetGroupRequest createDatasetGroupRequest = new CreateDatasetGroupRequest()

.withName("retail-dataset-group");

CreateDatasetGroupResult datasetGroupResult = personalizeClient.createDatasetGroup(createDatasetGroupRequest);

String datasetGroupArn = datasetGroupResult.getDatasetGroupArn();

// Create schema for interactions dataset

CreateSchemaRequest createSchemaRequest = new CreateSchemaRequest()

.withName("retail-interactions-schema")

.withSchema("{

\"type\": \"record\",

\"name\": \"Interactions\",

\"namespace\": \"com.amazonaws.personalize.schema\",

\"fields\": [

{\"name\": \"user\_id\", \"type\": \"string\"},

{\"name\": \"item\_id\", \"type\": \"string\"},

{\"name\": \"timestamp\", \"type\": \"long\"},

{\"name\": \"interaction\_type\", \"type\": \"string\"}

],

\"version\": \"1.0\"

}");

CreateSchemaResult schemaResult = personalizeClient.createSchema(createSchemaRequest);

String schemaArn = schemaResult.getSchemaArn();

// Create interactions dataset

CreateDatasetRequest createDatasetRequest = new CreateDatasetRequest()

.withName("retail-interactions-dataset")

.withDatasetType("INTERACTIONS")

.withDatasetGroupArn(datasetGroupArn)

.withSchemaArn(schemaArn);

personalizeClient.createDataset(createDatasetRequest);

// Add dataset import job (assuming dataset already available in S3)

CreateDatasetImportJobRequest createDatasetImportJobRequest = new CreateDatasetImportJobRequest()

.withJobName("retail-interactions-import-job")

.withDatasetArn("arn:aws:personalize:us-west-2:xxxxxx:dataset/retail-interactions-dataset")

.withDataSource(new DataSource().withDataLocation("s3://processed-interaction-data/output/"))

.withRoleArn("arn:aws:iam::xxxxxx:role/PersonalizeRole");

personalizeClient.createDatasetImportJob(createDatasetImportJobRequest);

// Create solution

CreateSolutionRequest createSolutionRequest = new CreateSolutionRequest()

.withName("retail-solution")

.withDatasetGroupArn(datasetGroupArn)

.withRecipeArn("arn:aws:personalize:::recipe/aws-user-personalization");

CreateSolutionResult solutionResult = personalizeClient.createSolution(createSolutionRequest);

String solutionArn = solutionResult.getSolutionArn();

// Create solution version

CreateSolutionVersionRequest createSolutionVersionRequest = new CreateSolutionVersionRequest()

.withSolutionArn(solutionArn);

personalizeClient.createSolutionVersion(createSolutionVersionRequest);

// Finally, create a campaign

CreateCampaignRequest createCampaignRequest = new CreateCampaignRequest()

.withName("retail-recommendation-campaign")

.withSolutionVersionArn("arn:aws:personalize:us-west-2:xxxxxx:solution/retail-solution/x.x.x")

.withMinProvisionedTPS(1);

personalizeClient.createCampaign(createCampaignRequest);

}

public static void main(String[] args) {

PersonalizeSetup setup = new PersonalizeSetup();

setup.createPersonalizeResources();

}

}

```

#### 3. Java Application Development

Use Amazon Personalize to fetch recommendations based on user queries.

\*\*Maven Dependencies (pom.xml)\*\*:

```xml

<dependencies>

<!-- AWS SDK dependencies -->

<dependency>

<groupId>com.amazonaws</groupId>

<artifactId>aws-java-sdk-personalize</artifactId>

<version>1.11.1034</version>

</dependency>

<dependency>

<groupId>com.amazonaws</groupId>

<artifactId>aws-java-sdk-cloudwatch</artifactId>

<version>1.11.1034</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.30</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>1.7.30</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.12.3</version>

</dependency>

</dependencies>

```

\*\*Java Code for Recommendations Using Amazon Personalize\*\*:

```java

import com.amazonaws.auth.AWSStaticCredentialsProvider;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;

import com.amazonaws.regions.Regions;

import com.amazonaws.services.personalize.AmazonPersonalize;

import com.amazonaws.services.personalize.AmazonPersonalizeClientBuilder;

import com.amazonaws.services.personalize.model.GetRecommendationsRequest;

import com.amazonaws.services.personalize.model.GetRecommendationsResult;

import com.amazonaws.services.personalize.model.Prediction;

import java.util.List;

import java.util.stream.Collectors;

public class PersonalizeService {

private final AmazonPersonalize personalizeClient;

private final String campaignArn = "arn:aws:personalize:us-west-2:xxxxxx:campaign/retail-recommendation-campaign";

public PersonalizeService() {

this.personalizeClient = AmazonPersonalizeClientBuilder.standard()

.withRegion(Regions.US\_WEST\_2)

.withCredentials(new AWSStaticCredentialsProvider(new ProfileCredentialsProvider().getCredentials()))

.build();

}

public List<String> getRecommendations(String userId) {

GetRecommendationsRequest request = new GetRecommendationsRequest()

.withCampaignArn(campaignArn)

.withUserId(userId);

GetRecommendationsResult result = personalizeClient.getRecommendations(request);

List<Prediction> predictions = result.getItemList();

return predictions.stream().map(Prediction::getItemId).collect(Collectors.toList());

}

public static void main(String[] args) {

PersonalizeService service = new PersonalizeService();

List<String> recommendations = service.getRecommendations("user123");

recommendations.forEach(System.out::println);

}

}

```

#### 4. Monitoring with Amazon CloudWatch

\*\*Java Code to Publish Custom Metrics to CloudWatch\*\*:

```java

import com.amazonaws.services.cloudwatch.AmazonCloudWatch;

import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;

import com.amazonaws.services.cloudwatch.model.Dimension;

import com.amazonaws.services.cloudwatch.model.MetricDatum;

import com.amazonaws.services.cloudwatch.model.PutMetricDataRequest;

import com.amazonaws.services.cloudwatch.model.StandardUnit;

public class CloudWatchMetrics {

private final AmazonCloudWatch cloudWatchClient;

public CloudWatchMetrics() {

this.cloudWatchClient = AmazonCloudWatchClientBuilder.defaultClient();

}

public void publishMetric(String metricName, double value) {

MetricDatum datum = new MetricDatum()

.withMetricName(metricName)

.withUnit(StandardUnit.Count)

.withValue(value)

.withDimensions(new Dimension().withName("AppName").withValue("RetailSearchApp"));

PutMetricDataRequest request = new PutMetricDataRequest()

.withNamespace("RetailSearchAppMetrics")

.withMetricData(datum);

cloudWatchClient.putMetricData(request);

}

public static void main(String[] args) {

CloudWatchMetrics metrics = new CloudWatchMetrics();

metrics.publishMetric("UserSearchQueries", 1);

}

}

```

### Integration and Full Application

Integrate the components to form a coherent retail recommendation application.

\*\*Main Application Code\*\*:

```java

public class Main {

public static void main(String[] args) {

// Initialize services

PersonalizeService personalizeService = new PersonalizeService();

CloudWatchMetrics cloudWatchMetrics = new CloudWatchMetrics();

// Simulate a user search query or interaction

String userId = "user123";

List<String> recommendations = personalizeService.getRecommendations(userId);

// Output recommendations

System.out.println("Top Recommendations:");

recommendations.forEach(System.out::println);

// Publish a custom metric to CloudWatch

cloudWatchMetrics.publishMetric("UserSearchQueries", 1);

}

}

```

### Conclusion

This project demonstrates how to leverage AWS Personalize within a Java application to generate product recommendations for a retail application. The setup involves preparing data with AWS Glue, setting up and training using Amazon Personalize, and finally integrating the services into a Java application, while monitoring the system performance with CloudWatch. You can further extend this project by including additional features, more sophisticated recommendation algorithms, and integrations with other AWS services as needed.