A diagram of data processing process

Description automatically generated

This is is a Big Data Engineering Project on Azure. Data was sourced from 2 data sources and ingested into the Data Lake in Azure. In the Data Lake, after a couple of data transformation and Machine Learning Model running, the result was saved down to Azure SQL database, and then connect it to Power BI for visualizing the data. Lastly, I used Azure Dev Ops to deploy CI/CD for the project.

**1/ Stage 1:** use Azure data factory to ingest data from 2 data sources, a postgres database on RDS and a WeCloudData's public Azure storage container.

. Step 0: : create separate resource group for the project so that we can delete the resource group after completing the project to avoid being further charged.

. Step 1: create a storage account, a container, and folders to accept the files from the data sources. It’s noted to create a data lake instead of a simple blob storage by ticking on the **Enable hierarchical namespace**.

A screenshot of a phone

Description automatically generated

The folder **postTypes** accepts the table **postTypes** on PostgreSQL; The folder **Users** accepts the table **Users** on PostgreSQL;  The folder **Posts** accepts today's Parquet files, only today's file, which means we need to empty the folder every day and import today's file from WecloudData's blob storage.

. Step 2: create a Data Factory Resource with 2 pipelines: copyOnceWeek and copyPostsEveryday

. Step 3: prior to create the 2 pipelines, we need to create 3 linked services

Linked Service 1: This linked service is used to link the RDS PostgreSQL database; we can call it ls\_rds\_pg. The linked service is used for pipeline 1. The PostgreSQL information is:



Host: de-rds.czm23kqmbd6o.ca-central-1.rds.amazonaws.com

Master username: postgres

Password: weclouddatade

Database: stack

Port: 5432

Psql on the EC2 of Airbyte

Linked Service 2: This linked service is used to link WeCloudData's public storage blob; we can call it ls\_wcd\_blob. The storage account access key is: HMAYNz/35GY7nWarOjv981XVPCE5LK86y4mZtGS2OYB7Ks87gMwuiL0PyY0vsURFYQ8C0xK+5VuH+ASt0i2hZw==. The files are under the storage account wcddestorageexternal. The linked service is used for pipeline 2.



Linked Service 3: This linked service is used to link the storage blob; we can call it ls\_my\_blob. The linked service is used for both pipelines.



. Step 4: create 6 datasets:

For pipeline 1, we need to create 4 datasets:

Two datasets represent the postTypes and Users tables on PostgreSQL.

Two datasets represent the postTypes and Users files copied from PostgreSQL and stored in our blob storage.

 For pipeline 2, you need to create 2 datasets:

A dataset represents the Posts Parquet files on WeCloudData's storage blob.

A dataset represents the Posts Parquet files on your storage blob.

. Step 5: create pipeline 1: copyOnceWeek

A screenshot of a computer

Description automatically generated

CleanFolder: to empty the postTypes and Users folders prior to copying the txt files.

Set the trigger for the pipeline to run once per week

. Step 6: create pipeline 2: copyPostsEveryday

A screenshot of a computer

Description automatically generated

CleanFolder: to empty the Posts folder prior to copying the parquet files.

Set the trigger for the pipeline to run once per day

**2/ Stage2:**

**. Step 1**: mount Azure storage to Azure Databricks

. Create Databricks workspace, then compute, and notebook

. In Azure, allow the Azure Databricks to access the storage container by follow the below steps

Search for Microsoft Entra ID

Go to App registrations, and click New registration

On the new page, give it a name like databricksaccess. Note down the application ID and Directory ID

A screenshot of a computer

Description automatically generated

Then go to Certificates and secrets, and click on new clients secret

A screenshot of a computer

Description automatically generated

Note down the value of client secret

A screenshot of a computer

Description automatically generated

Then go to the storage account Access Control (IAM) , then add role assignment

A screenshot of a computer

Description automatically generated

Then search for Storage Blob Data Contributor, and click next, and then click on select member to add the databricksaccess in

A screenshot of a computer

Description automatically generated

Below link is detail guidance of these steps

[Access storage with Microsoft Entra ID (formerly Azure Active Directory) using a service principal | Databricks on AWS](https://docs.databricks.com/en/connect/storage/aad-storage-service-principal.html)

. On the azure notebook, run the below code to mount storage container to azure databricks directory

storageAccountName = '………….'

containerName = '……………….'

applicationId = …………………………………………………………'

directoryID = ‘………………………………………………….’

secretValue = ‘……………………………………………………………………’

endpoint = 'https://login.microsoftonline.com/' + directoryID + '/oauth2/token'

source = 'abfss://' + containerName + '@' + storageAccountName + '.dfs.core.windows.net/'

mountPoint = "/mnt/Phase2Project"

configs = {"fs.azure.account.auth.type": "OAuth",

"fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",

"fs.azure.account.oauth2.client.id": applicationId,

"fs.azure.account.oauth2.client.secret": secretValue,

"fs.azure.account.oauth2.client.endpoint": endpoint}

dbutils.fs.mount(source = source,mount\_point = mountPoint, extra\_configs = configs)

display(dbutils.fs.ls(“/mnt/Phase2Project”)

. **Step 2**: NLP to train a machine learning model. The model will use everydayCopy Posts (parquet files in the mnt\Posts folder) and copyonceWeek(postTypes.txt in the mnt\postTypes folder) to tell us what each post is about (the topic of the post).

* Join the Posts and posttypes tables, because we need to use the Type column in the posttypes table to filter out the posts in Posts table we want. And filter the data we want.
* Prepare the training data for the machine learning training.
* Train the machine learning model;
* save the model to a Azure storage folder so that we can use it in Step 3.

# Creating a spark session

from pyspark.sql import SparkSession

spark = (SparkSession

         .builder

         .appName("Table Loading")

         .getOrCreate())

sc = spark.sparkContext

#create Posts dataframe

# File location -- recall our mount storage workshop, data was mounted into '/mnt/deBDProject'

file\_location = "/mnt/Phase2Project/Posts/\*"

posts = spark.read.parquet(file\_location)

display(posts)

#create postTypes dataframe

from pyspark.sql.types import \*

schema = StructType([StructField("id", IntegerType(), True), StructField("Type", StringType(), True)])

file\_location = "/mnt/Phase2Project/postTypes/\*.txt"

postType = (spark.read.option("header", "true").option("sep", ",").schema(schema).csv(file\_location))

display(postType)

A screenshot of a computer

Description automatically generated

#create users dataframe

# Creating the schema for the users table

from pyspark.sql.types import \*

users\_schema = StructType([

    StructField("id", IntegerType(), True),

    StructField("Age", IntegerType(), True),

    StructField("CreationDate", DateType(), True),

    StructField("DisplayName", StringType(), True),

    StructField("DownVotes", IntegerType(), True),

    StructField("EmailHash", StringType(), True),

    StructField("Location", StringType(), True),

    StructField("Reputation", IntegerType(), True),

    StructField("UpVotes", IntegerType(), True),

    StructField("Views", IntegerType(), True),

    StructField("WebsiteUrl", StringType(), True),

    StructField("AccountId", IntegerType(), True)

])

# Creating the users dataframe

file\_location = "/mnt/Phase2Project/ml\_training/users.csv"

users = (spark.read.option("header", "true").option("sep", ",").schema(users\_schema).csv(file\_location))

display(users)

A screenshot of a computer

Description automatically generated

# Save the 3 tables to databricks local file system

posts.write.parquet("/tmp/project/posts.parquet", mode='overwrite')

postType.write.parquet("/tmp/project/PostType.parquet", mode='overwrite')

users.write.parquet("/tmp/project/user.parquet", mode='overwrite')

from pyspark.sql import SparkSession

from pyspark.sql.functions import \*

# Creating Spark Session

spark = (SparkSession.builder.appName("ML Model").getOrCreate())

sc = spark.sparkContext

# Read in the tables

posts = spark.read.parquet("/tmp/project/posts.parquet")

postType = spark.read.parquet("/tmp/project/PostType.parquet")

users = spark.read.parquet("/tmp/project/user.parquet")

# at this moment, we only use Posts and posttypes to train the model. so let's join them with the posttype id.

df = posts.join(postType, posts.PostTypeId == postType.id)

display(df)

A screenshot of a computer

Description automatically generated

# Filter the dataframe to only include questions

df = df.filter(col("Type") == "Question")

display(df)

A screenshot of a computer

Description automatically generated

# Formatting the 'Body' and `Tag` columns for machine learning training

df = (df.withColumn('Body', regexp\_replace(df.Body, r'<.\*?>', '')) # Transforming HTML code to strings

      .withColumn("Tags", split(trim(translate(col("Tags"), "<>", " ")), " ")) # Making a list of the tags

)

display(df)

A screenshot of a computer

Description automatically generated

df = df.select(col("Body").alias("text"), col("Tags"))

display(df)

A screenshot of a computer

Description automatically generated

# Select the "text" column and explode the "Tags" column

df = df.select("text", explode("Tags").alias("tags"))

#reduce the dataframe from 2,498 rows to 2000 rows so that the output does not exceed the limit

df = df.limit(2000)

# Display the resulting DataFrame

display(df)

A screenshot of a computer

Description automatically generated

# saving the file as a checkpoint (in case the cluster gets terminated)

df.write.parquet("/tmp/project/df.parquet",mode='overwrite')

# Saving the dataframe to memory for repetitive use

df.cache()

df.count()

# Preprocessing the data

cleaned = df.withColumn('text', regexp\_replace('text', r"http\S+", "")) \

                    .withColumn('text', regexp\_replace('text', r"[^a-zA-z]", " ")) \

                    .withColumn('text', regexp\_replace('text', r"\s+", " ")) \

                    .withColumn('text', lower('text')) \

                    .withColumn('text', trim('text'))

display(cleaned)

A screenshot of a computer

Description automatically generated

from pyspark.ml.feature import Tokenizer

tokenizer = Tokenizer(inputCol="text", outputCol="tokens")

tokenized = tokenizer.transform(cleaned)

display(tokenized)

A screenshot of a computer

Description automatically generated

#Stopword Removal

from pyspark.ml.feature import StopWordsRemover

stopword\_remover = StopWordsRemover(inputCol="tokens", outputCol="filtered")

stopword = stopword\_remover.transform(tokenized)

display(stopword)

A screenshot of a computer

Description automatically generated

#CountVectorizer (TF - Term Frequency)

from pyspark.ml.feature import CountVectorizer

cv = CountVectorizer(vocabSize=2\*\*16, inputCol="filtered", outputCol='cv')

cv\_model = cv.fit(stopword)

text\_cv = cv\_model.transform(stopword)

display(text\_cv)

A screenshot of a computer

Description automatically generated

#TF-IDF Vectorization

from pyspark.ml.feature import HashingTF, IDF

idf = IDF(inputCol='cv', outputCol="features", minDocFreq=5) #minDocFreq: remove sparse terms

idf\_model = idf.fit(text\_cv)

text\_idf = idf\_model.transform(text\_cv)

display(text\_idf)

A screenshot of a computer

Description automatically generated

#label encoding

from pyspark.ml.feature import StringIndexer

label\_encoder = StringIndexer(inputCol = "tags", outputCol = "label")

le\_model = label\_encoder.fit(text\_idf)

final = le\_model.transform(text\_idf)

display(final)

A screenshot of a computer

Description automatically generated

#model training

from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression(maxIter=100)

lr\_model = lr.fit(final)

predictions = lr\_model.transform(final)

display(predictions)

A screenshot of a computer

Description automatically generated

#model evaluation

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = MulticlassClassificationEvaluator(predictionCol="prediction")

roc\_auc = evaluator.evaluate(predictions)

accuracy = predictions.filter(predictions.label == predictions.prediction).count() / float(predictions.count())

print("Accuracy Score: {0:.4f}".format(accuracy))

print("ROC-AUC: {0:.4f}".format(roc\_auc))

#create pipeline

# Importing all the libraries

from pyspark.sql.functions import split, translate, trim, explode, regexp\_replace, col, lower

from pyspark.ml.feature import Tokenizer, StopWordsRemover, HashingTF, IDF, StringIndexer

from pyspark.ml.classification import LogisticRegression

from pyspark.ml import Pipeline

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Preparing the data

# Step 1: Creating the joined table

df = posts.join(postType, posts.PostTypeId == postType.id)

# Step 2: Selecting only Question posts

df = df.filter(col("Type") == "Question")

# Step 3: Formatting the raw data

df = (df.withColumn('Body', regexp\_replace(df.Body, r'<.\*?>', ''))

      .withColumn("Tags", split(trim(translate(col("Tags"), "<>", " ")), " "))

)

# Step 4: Selecting the columns

df = df.select(col("Body").alias("text"), col("Tags"))

# Step 5: Getting the tags

df = df.select("text", explode("Tags").alias("tags"))

# Step 6: Clean the text

cleaned = df.withColumn('text', regexp\_replace('text', r"http\S+", "")) \

                    .withColumn('text', regexp\_replace('text', r"[^a-zA-z]", " ")) \

                    .withColumn('text', regexp\_replace('text', r"\s+", " ")) \

                    .withColumn('text', lower('text')) \

                    .withColumn('text', trim('text'))

# Machine Learning

# Step 1: Train Test Split

train, test = cleaned.randomSplit([0.9, 0.1], seed=20200819)

# Step 2: Initializing the transfomers

tokenizer = Tokenizer(inputCol="text", outputCol="tokens")

stopword\_remover = StopWordsRemover(inputCol="tokens", outputCol="filtered")

cv = CountVectorizer(vocabSize=2\*\*16, inputCol="filtered", outputCol='cv')

idf = IDF(inputCol='cv', outputCol="features", minDocFreq=5)

label\_encoder = StringIndexer(inputCol = "tags", outputCol = "label")

lr = LogisticRegression(maxIter=100)

# Step 3: Creating the pipeline

pipeline = Pipeline(stages=[tokenizer, stopword\_remover, cv, idf, label\_encoder, lr])

# Step 4: Fitting and transforming (predicting) using the pipeline

pipeline\_model = pipeline.fit(train)

predictions = pipeline\_model.transform(test)

# Saving model object to the /mnt/deBDProject directory.

pipeline\_model.write().overwrite().save('/mnt/Phase2Project/model')

# Save the the String Indexer to decode the encoding. We need it in the future Sentiment Analysis.

le\_model.write().overwrite().save('/mnt/Phase2Project/stringindexer')

**Step 3:** Sentiment Analysis

from pyspark.sql.functions import \*

from pyspark.sql import SparkSession

spark = (SparkSession

         .builder

         .appName("ML Model")

         .getOrCreate())

sc = spark.sparkContext

#Load the ML model and the stringIdexer generated in the previous workshop, and also the Posts file

posts = spark.read.parquet("/mnt/Phase2Project/Posts/\*")

ml\_model = "/mnt/Phase2Project/model"

stringindexer = "/mnt/Phase2Project/stringindexer"

#Create a UDF (User Defined Function) function to filter and transform the data and generate model result.

def predictions\_udf(df, ml\_model, stringindexer):

    from pyspark.sql.functions import col, regexp\_replace, lower, trim

    from pyspark.ml import PipelineModel

    # Filter out empty body text

    df = df.filter("Body is not null")

    # Making sure the naming of the columns are consistent with the model

    df = df.select(col("Body").alias("text"), col("Tags"))

    # Preprocessing of the feature column

    cleaned = df.withColumn('text', regexp\_replace('text', r"http\S+", "")) \

                    .withColumn('text', regexp\_replace('text', r"[^a-zA-z]", " ")) \

                    .withColumn('text', regexp\_replace('text', r"\s+", " ")) \

                    .withColumn('text', lower('text')) \

                    .withColumn('text', trim('text'))

    # Load in the saved pipeline model

    model = PipelineModel.load(ml\_model)

    # Making the prediction

    prediction = model.transform(df)

    predicted = prediction.select(col('text'), col('Tags'), col('prediction'))

    # Decoding the indexer

    from pyspark.ml.feature import StringIndexerModel, IndexToString

    # Load in the StringIndexer that was saved

    indexer = StringIndexerModel.load(stringindexer)

    # Initialize the IndexToString converter

    i2s = IndexToString(inputCol = 'prediction', outputCol = 'decoded', labels = indexer.labels)

    converted = i2s.transform(predicted)

    # Display the important columns

    return converted

result = predictions\_udf(posts,ml\_model, stringindexer)

# change the column name

topics = result.withColumnRenamed('decoded', 'topic').select('topic')

# Aggregate the topics and calculate the total qty of each topic

topic\_qty = topics.groupBy(col("topic")).agg(count('topic').alias('qty')).orderBy(desc('qty'))

# define this function

def crt\_sgl\_file(result\_path):

        # write the result to a folder container several files

        path = "/mnt/Phase2Project/BI/ml\_result"

        topic\_qty.write.option("delimiter", ",").option("header", "true").mode("overwrite").csv(path)

        # list the folder, find the csv file

        filenames = dbutils.fs.ls(path)

        name = ''

        for filename in filenames:

            if filename.name.endswith('csv'):

                org\_name = filename.name

        # copy the csv file to the path you want to save, in this example, we use  "/mnt/Phase2Project/BI/ml\_result.csv"

        dbutils.fs.cp(path + '/'+ org\_name, result\_path)

        # delete the folder

        dbutils.fs.rm(path, True)

        print('single file created')

# run the function

result\_path = "/mnt/Phase2Project/BI/ml\_result.csv"

crt\_sgl\_file(result\_path)

**Stage 3**: rather than visualizing data by Azure Synapse (as it’s looked pretty simple), I switched to copy the result to Azure SQL Database, and then connect to Power BI for visualization.

. Step 1: create Azure SQL Database, and then create table namely ML\_Result with the schema matched to that of ml\_result.csv

result\_path = "/mnt/Phase2Project/BI/ml\_result.csv"

CREATE TABLE ML\_result (

topic VARCHAR(MAX),

qty INT

);

. Step 2: create a copy activity with 1 service link to Azure SQL Database (no need for the the blob container as it’s already created). 2 dataset, 1 for the source ( result\_path = "/mnt/Phase2Project/BI/ml\_result.csv"), and 1 for the Azure SQL Database

A screenshot of a computer

Description automatically generated

. Step 3: connect power BI and Azure SQL Database

In the server name, please make sure to add server name, and the string in red phase2-kha.database.windows.net

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

. Step 4: visualization

A screen shot of a graph

Description automatically generated

A screenshot of a computer screen

Description automatically generated