M7T3 – Python libraries for rich data collection practical activities

Practical Lab Activities (2 Hours)

Lab Activity 1: Hands-On Activity: Kafka-Python and Avro (30 minutes)

Objective: Utilise Python libraries such as kafka-python and Avro, and interact with schema registries.

Problem: Imagine your company, a financial services firm, needs to process and analyse user transaction data in real-time to detect fraudulent activities. Your task is to set up a Kafka environment and use Python to produce and consume user data in Avro format.

Step-by-step instructions

Setup:

- **Install Kafka**: Ensure Kafka is installed and running on your machine.
- **Install kafka-python**: Run pip install kafka-python to install the kafka-python library.

Tasks:

Schema Definition:

- **Define an Avro Schema**: Create a file named user.avsc with the following content:{
  "namespace": "example.avro",
  "type": "record",
  "name": "User",
  "fields": [
  {"name": "username", "type": "string"},
  {"name": "email", "type": ["null", "string"], "default": null},
  {"name": "age", "type": "int"}

```
 ]
}
```

Serialization:

- **Install avro-python3**: Run pip install avro-python3.
- **Serialize Data**: Use the following code to serialize data:from avro.io import DatumWriter, BinaryEncoder
  from avro.schema import Parse
  import io

  schema = Parse(open("user.avsc", "r").read())
  writer = DatumWriter(schema)
  bytes_writer = io.BytesIO()
  encoder = BinaryEncoder(bytes_writer)
  user_data = {"username": "john_doe", "email": "john@example.com", "age": 30}
  writer.write(user_data, encoder)
  raw_bytes = bytes_writer.getvalue()

Send Data to Kafka:

- **Produce Messages**: Use the following code to send serialized data to a Kafka topic:from kafka import KafkaProducer

  producer = KafkaProducer(bootstrap_servers='localhost:9092')
  producer.send('users', raw_bytes)
  producer.flush()

Deserialization:

- **Consume Messages**: Use the following code to consume and deserialize data from a Kafka topic:from kafka import KafkaConsumer
  from avro.io import DatumReader, BinaryDecoder
  import io

```
consumer = KafkaConsumer('users', bootstrap_servers='localhost:9092')
reader = DatumReader(schema)

for message in consumer:
    bytes_reader = io.BytesIO(message.value)
    decoder = BinaryDecoder(bytes_reader)
    user = reader.read(decoder)
    print(f"Received message: {user}")
```

Expected Outcomes:

- **Kafka Setup**: Kafka is installed and running on your machine.
- **Schema Definition**: An Avro schema (user.avsc) is created and correctly defines the user data structure.
- **Data Serialization**: User data is successfully serialized using the Avro schema.
- **Data Production**: Serialized user data is sent to a Kafka topic.
- **Data Consumption**: Serialized user data is consumed from the Kafka topic and deserialized back into its original form.

Sample Data:

- **Before Serialization**:{
  "username": "john_doe",
  "email": "john@example.com",
  "age": 30
  }

- **After Serialization:** b'\x06john_doe\x1cjohn@example.com <'

Lab Activity 2: Scrapy for Data Collection (30 minutes)

Objective: Integrate Scrapy for data collection from web sources.

Problem: Imagine you now work for an e-commerce platform that needs to collect product reviews from various websites to analyse customer sentiment. You need to use Scrapy to collect data and send it to a Kafka topic for further processing.

Step-by-step instructions

Setup:

- **Install Scrapy**: Run pip install scrapy.

Tasks:

Creating a New Project:

- **Create Project**: Run scrapy startproject myproject to create a new Scrapy project.

Defining a Spider:

- **Create Spider**: Navigate to the spiders directory and create a file named quotes_spider.py with the following content:import scrapy

  class QuotesSpider(scrapy.Spider):
  name = "quotes"
  start_urls = ['http://quotes.toscrape.com/']

  def parse(self, response):
  for quote in response.css('div.quote'):
  yield {
  'text': quote.css('span.text::text').get(),
  'author': quote.css('small.author::text').get(),
  }

Running the Spider:

- **Run Spider**: Run scrapy crawl quotes -o quotes.json to execute the spider and output the scraped data to a JSON file.

Integration with Kafka:

- **Send Data to Kafka**: Modify the parse method to send the scraped data to a Kafka topic:from kafka import KafkaProducer
  import json

  producer = KafkaProducer(bootstrap_servers='localhost:9092',

```
    value_serializer=lambda v: json.dumps(v).encode('utf-8'))

    class QuotesSpider(scrapy.Spider):
        name = "quotes"
        start_urls = ['http://quotes.toscrape.com/']

        def parse(self, response):
            for quote in response.css('div.quote'):
                item = {
                    'text': quote.css('span.text::text').get(),
                    'author': quote.css('small.author::text').get(),
                }
                producer.send('quotes', item)
```

Expected Outcomes:

- **Scrapy Setup**: Scrapy is installed, and a new project is created.
- **Spider Definition**: A Scrapy spider is defined to scrape data from a specified website.
- **Data Collection**: The spider successfully collects data and outputs it to a JSON file.
- **Kafka Integration**: The collected data is sent to a Kafka topic for further processing.

Sample Data:

- **Before Scraping**:
    "The greatest glory in living lies not in never falling, but in rising every time we fall."
    <small class="author">Nelson Mandela</small>

- **After Scraping**:{
    "text": "The greatest glory in living lies not in never falling, but in rising every time we fall.",
    "author": "Nelson Mandela"

}

Lab Activity 3: Data Validation and Processing (20 minutes)

Objective: Implement data validation and processing using schema registries.

Problem: Now imagine you work for a healthcare provider that needs to ensure the integrity and consistency of patient data. Your task is to validate and process user data using a schema registry.

Step-by-step instructions

Setup:

- **Install Confluent Schema Registry**: Follow the Confluent Schema Registry installation guide.
- **Start Schema Registry**: Run the Schema Registry using the command:bin/schema-registry-start etc/schema-registry/schema-registry.properties

Tasks:

Validation Rules:

- **Define Schema**: Use the previously defined Avro schema (user.avsc).
- **Register Schema**: Register the schema using the REST API:curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
  --data '{"schema": "{\"type\": \"record\", \"name\": \"User\", \"fields\": [{\"name\": \"username\", \"type\": \"string\"}, {\"name\": \"email\", \"type\": [\"null\", \"string\"], \"default\": null}, {\"name\": \"age\", \"type\": \"int\"}]}"}' \
  http://localhost:8081/subjects/User-value/versions

Validation Implementation:

- **Validate Data**: Modify the producer code to include validation logic:import re

  def validate_user_data(user_data):
  if not re.match(r"[^@]+@[^@]+\.[^@]+", user_data['email']):

```
            raise ValueError("Invalid email format")
        if len(user_data['password']) < 8:
            raise ValueError("Password must be at least 8 characters long")
        if user_data['age'] < 13:
            raise ValueError("User must be at least 13 years old")
        return True

    try:
        validate_user_data(user_data)
        producer.produce(topic='user_registrations', value=user_data)
    except ValueError as e:
        log.error(f"Validation error: {e}")
```

Expected Outcomes:

- **Schema Registry Setup**: Confluent Schema Registry is installed and running.
- **Schema Registration**: The Avro schema is registered with the schema registry.
- **Data Validation**: User data is validated against the schema before being sent to Kafka.
- **Error Handling**: Validation errors are correctly identified and logged.

Sample Data:

- **Before Validation**:{
  "username": "john_doe",
  "email": "john@example.com",
  "age": 30
  }

- **After Validation**:{
  " {
  "username": "john_doe",
  "email": "john@example.com",

```
  "age": 30
}
```