1. Introdução

Este documento detalha o fluxo completo de um sistema de processamento de invoices, desde a recepção dos dados até sua análise. O sistema utiliza tecnologias modernas como FastAPI, Kafka e possivelmente análise de sentimentos para processar críticas de clientes.

2. Estrutura do Invoice

O invoice é a unidade básica de informação no sistema, contendo os seguintes campos:

* InvoiceNo: Número único de identificação do invoice.
* MeioEnvio: Meio pelo qual o invoice foi enviado (ex: celular, totem, computador).
* Description: Descrição da crítica do cliente, podendo ser positiva ou negativa.
* InvoiceDate: Data e hora do registro do invoice.
* CustomerID: Identificador único do cliente.
* City: Cidade do cliente.
* Country: País do cliente.

Exemplo de invoice em formato JSON:

{

"InvoiceNo": 536365,

"MeioEnvio": "IOS13",

"Description": "App lento",

"InvoiceDate": "12/02/2024 8:26",

"CustomerID": 17850,

"City": "Sao Paulo",

"Country": "Brasil"

}

### Estrutura do Criticidade: Estrutura do Criticidade

O documento de criticidade é a unidade básica de informação no sistema para armazenar as críticas dos clientes, contendo os seguintes campos:

* **InvoiceNo:** Número único de identificação do invoice.
* **MeioEnvio:** Meio pelo qual o invoice foi enviado (ex: IOS13, totem, computador).
* **Description:** Descrição da crítica do cliente, podendo ser positiva ou negativa.
* **InvoiceDate:** Data e hora do registro do invoice.
* **CustomerID:** Identificador único do cliente.
* **City:** Cidade do cliente.
* **Country:** País do cliente.
* **SentimentRating:** Nota do sentimento atribuída à crítica (ex: 1 a 5).
* **Agrupador:** Categoria ou grupo ao qual a crítica pertence (ex: Aplicativo, Serviço).
* **Sentimento:** Qualificação do sentimento da crítica (ex: Positivo, Negativo).
* **nome\_da\_agencia:** Nome da agência responsável pela crítica após ETL e cruzamento de informações.
* **latitude:** Latitude da localização da agência.
* **longitude:** Longitude da localização da agência.
* **estado:** Estado do cliente.

### Exemplo de Criticidade em formato JSON:

json

Copiar código

{

"InvoiceNo": 536365,

"MeioEnvio": "IOS13",

"Description": "App lento",

"InvoiceDate": "12/02/2024 08:26",

"CustomerID": 49936,

"City": "São Paulo",

"Country": "Brasil",

"SentimentRating": 3,

"Agrupador": "Aplicativo",

"Sentimento": "Negativo",

"nome\_da\_agencia": "Sede",

"latitude": -23.590954,

"longitude": -46.690858,

"estado": "SP"

}

3. Recepção de Dados (API Ingest)

3.1 Localização do Código

O código responsável pela ingestão de dados está localizado em:

C:\Users\servidor\Downloads\dm2\document-streaming-main\API-Ingest\app\main.py

3.2 Tecnologias Utilizadas

O sistema utiliza FastAPI para criar uma API RESTful que recebe os invoices. As principais bibliotecas utilizadas são:

* FastAPI: Framework web para criar APIs com Python.
* Pydantic: Biblioteca para validação de dados e configurações.
* Kafka-Python: Cliente Kafka para Python.

3.3 Funcionamento da API

* A API expõe um endpoint POST para receber os invoices.
* Os dados recebidos são validados usando modelos Pydantic.
* Após a validação, os dados são enviados para um tópico Kafka.

4. Processamento de Dados

4.1 Kafka como Message Broker

* Os invoices são enviados para um tópico Kafka específico.
* Kafka atua como um buffer, permitindo que o sistema processe grandes volumes de dados de forma assíncrona.

4.2 Consumidor Kafka

* Um consumidor Kafka lê os dados do tópico.
* O consumidor pode estar implementado em um serviço separado.

5. Análise de Sentimentos

5.1 Processamento da Descrição

* O campo "Description" do invoice é analisado para determinar o sentimento (positivo ou negativo).
* Pode-se utilizar bibliotecas de processamento de linguagem natural (NLP) como NLTK ou spaCy.

5.2 Categorização

* Com base na análise, o sistema categoriza a crítica como positiva ou negativa.

6. Armazenamento e Visualização

6.1 Banco de Dados

* Os dados processados são armazenados em um banco de dados para análises futuras.
* Pode-se utilizar um banco de dados relacional (como PostgreSQL) ou não-relacional (como MongoDB).

6.2 Dashboard

* Um dashboard pode ser implementado para visualizar métricas e tendências.
* Ferramentas como Grafana ou Tableau podem ser utilizadas para criar visualizações.

7. Considerações de Segurança e Escalabilidade

7.1 Segurança

* Implementação de autenticação e autorização na API.
* Criptografia de dados sensíveis.

7.2 Escalabilidade

* Uso de containers Docker para facilitar a implantação e escalabilidade.
* Configuração de múltiplas instâncias do consumidor Kafka para processamento paralelo.

8. Monitoramento e Logging

8.1 Logs

* Implementação de logging detalhado em todas as etapas do processo.

8.2 Alertas

* Configuração de alertas para erros críticos ou anomalias nos dados.

9. Conclusão

Este sistema oferece uma solução robusta e escalável para processar críticas de clientes em tempo real, permitindo análises rápidas e tomadas de decisão baseadas em dados.

Título: Código de Ingestão de Invoices via API FastAPI

1. Importações

from fastapi import FastAPI, status, HTTPException

from fastapi.encoders import jsonable\_encoder

from fastapi.responses import JSONResponse

import json

from pydantic import BaseModel

from datetime import datetime

from kafka import KafkaProducer, errors

Explicação:

* FastAPI: Framework web para criar APIs RESTful de forma rápida e eficiente.
* Pydantic: Utilizada para validação de dados e serialização.
* datetime: Para manipulação de datas e horas.
* KafkaProducer: Cliente Kafka para envio de mensagens.

2. Definição do Modelo de Dados

class InvoiceItem(BaseModel):

InvoiceNo: int

MeioEnvio: str

Description: str

InvoiceDate: str

CustomerID: int

City: str

Country: str

Explicação:

* Define a estrutura do invoice usando Pydantic.
* Garante a validação dos tipos de dados recebidos.

3. Inicialização da Aplicação FastAPI

app = FastAPI()

Explicação:

* Cria uma instância da aplicação FastAPI.

4. Rota Raiz

@app.get("/")

async def root():

return {"message": "Hello World"}

Explicação:

* Rota básica para verificar se a API está funcionando.

5. Rota de Postagem de Invoice

@app.post("/invoiceitem")

async def post\_invoice\_item(item: InvoiceItem):

# ... (código omitido para brevidade)

Explicação:

* Endpoint principal para receber novos invoices.
* Usa o modelo InvoiceItem para validação automática dos dados.

6. Processamento do Invoice

try:

date = datetime.strptime(item.InvoiceDate, "%d/%m/%Y %H:%M")

item.InvoiceDate = date.strftime("%d-%m-%Y %H:%M:%S")

json\_of\_item = jsonable\_encoder(item)

json\_as\_string = json.dumps(json\_of\_item)

produce\_kafka\_string(json\_as\_string)

return JSONResponse(content=json\_of\_item, status\_code=201)

Explicação:

* Converte a data para um formato padrão.
* Serializa o item para JSON.
* Envia o JSON para o Kafka.
* Retorna o item criado com status 201 (Created).

7. Tratamento de Erros

except ValueError:

return JSONResponse(content=jsonable\_encoder(item), status\_code=400)

except errors.NoBrokersAvailable as e:

print("Kafka broker not available:", e)

raise HTTPException(status\_code=500, detail="Kafka broker not available")

Explicação:

* Trata erros de formato de data (ValueError).
* Trata erros de conexão com o Kafka.

8. Função de Produção para o Kafka

def produce\_kafka\_string(json\_as\_string):

try:

producer = KafkaProducer(bootstrap\_servers='kafka:9092', acks=1)

producer.send('ingestion-topic', bytes(json\_as\_string, 'utf-8'))

producer.flush()

except errors.NoBrokersAvailable as e:

print("Kafka broker not available:", e)

raise

Explicação:

* Cria um produtor Kafka.
* Envia a mensagem para o tópico 'ingestion-topic'.
* Trata erros de conexão com o Kafka.

9. Inicialização do Servidor

if \_\_name\_\_ == "\_\_main\_\_":

print("Iniciando o servidor...")

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000)

print("Servidor iniciado.")

Explicação:

* Inicia o servidor usando Uvicorn quando o script é executado diretamente.
* Configura o servidor para escutar em todas as interfaces na porta 8000.

10. Considerações Finais

Este código implementa uma API robusta para ingestão de invoices, com as seguintes características:

* Validação de dados usando Pydantic.
* Tratamento de erros para formatos de data inválidos e problemas de conexão com o Kafka.
* Integração com Kafka para processamento assíncrono dos dados.
* Uso de FastAPI para criar uma API RESTful eficiente e fácil de mante

Título: Configuração e Uso do Kafka para Processamento de Invoices

1. Introdução ao Kafka

Kafka é uma plataforma de streaming distribuída que permite publicar, subscrever, armazenar e processar fluxos de registros em tempo real. É utilizada neste sistema para garantir um processamento assíncrono e escalável dos invoices.

2. Acesso ao Ambiente Kafka

Abrindo o kafka Atach Shell ou o terminal do Kafka (container)

cd opt/bitnami/kafka/bin/

Explicação:

* Este comando navega até o diretório onde estão localizados os scripts de administração do Kafka.
* É necessário acessar o shell do container Kafka para executar estes comandos.
* O uso de containers facilita a implantação e isolamento do ambiente Kafka.

3. Criação do Tópico Kafka

./kafka-topics.sh --create --topic ingestion-topic --bootstrap-server localhost:9092

Explicação:

* Este comando cria um novo tópico chamado "ingestion-topic".
* --create: Indica que estamos criando um novo tópico.
* --topic ingestion-topic: Define o nome do tópico.
* --bootstrap-server localhost:9092: Especifica o servidor Kafka a ser usado.
* O tópico é onde as mensagens (invoices) serão publicadas e consumidas.
* A criação de um tópico específico permite organizar e isolar diferentes fluxos de dados.

4. Inicialização do Consumidor Kafka

./kafka-console-consumer.sh --topic ingestion-topic --bootstrap-server localhost:9092

Explicação:

* Este comando inicia um consumidor Kafka que lê mensagens do tópico "ingestion-topic".
* --topic ingestion-topic: Especifica o tópico do qual consumir mensagens.
* --bootstrap-server localhost:9092: Define o servidor Kafka a ser usado.
* O consumidor ficará ativo, esperando por novas mensagens no tópico.
* Útil para debug e monitoramento em tempo real das mensagens recebidas.

5. Inicialização do Produtor Kafka (Opcional)

./kafka-console-producer.sh --topic ingestion-topic --bootstrap-server localhost:9092

Explicação:

* Este comando inicia um produtor Kafka que pode enviar mensagens para o tópico "ingestion-topic".
* Útil para testes manuais e debug do sistema.
* Permite enviar mensagens diretamente para o tópico, simulando o comportamento da API.

6. Fluxo de Dados

Após a configuração:

1. A API (FastAPI) envia mensagens para o tópico "ingestion-topic".
2. O consumidor Kafka, quando ativo, captura essas mensagens em tempo real.
3. As mensagens podem ser processadas por serviços adicionais conectados ao consumidor.

7. Considerações Importantes

7.1 Escalabilidade

* Kafka permite múltiplos produtores e consumidores, facilitando a escalabilidade horizontal.
* Pode-se adicionar mais consumidores para processar mensagens em paralelo.

7.2 Persistência

* Kafka armazena mensagens por um período configurável, permitindo reprocessamento se necessário.

7.3 Particionamento

* O tópico pode ser particionado para melhor distribuição e paralelismo no processamento.

7.4 Configurações Adicionais

* Retenção de mensagens, número de partições, e fator de replicação podem ser ajustados conforme necessidade.

8. Segurança

* Em ambientes de produção, é crucial configurar autenticação e autorização no Kafka.
* A comunicação pode ser criptografada usando SSL/TLS.

9. Monitoramento

* Ferramentas como Kafka Manager ou Confluent Control Center podem ser usadas para monitorar o cluster Kafka.
* Métricas importantes incluem latência, throughput, e offset lag dos consumidores.

10. Conclusão

Esta configuração do Kafka estabelece uma infraestrutura robusta para o processamento assíncrono de invoices. Ela permite que o sistema escale facilmente para lidar com grandes volumes de dados, mantendo a confiabilidade e a capacidade de recuperação em caso de falhas.

Título: Configuração do Fluxo de Dados do Kafka para o Spark Streaming

1. Introdução

Esta etapa do projeto envolve a criação de um novo tópico Kafka para facilitar a comunicação entre o sistema de ingestão de dados (API-Ingest) e o Spark Streaming. Este processo é crucial para estabelecer um pipeline de dados eficiente e escalável.

2. Criação do Novo Tópico Kafka

./kafka-topics.sh --create --topic spark-output --bootstrap-server localhost:9092

Explicação:

* Este comando cria um novo tópico Kafka chamado "spark-output".
* --create: Indica a criação de um novo tópico.
* --topic spark-output: Define o nome do novo tópico.
* --bootstrap-server localhost:9092: Especifica o servidor Kafka a ser utilizado.

Importância:

* O tópico "spark-output" serve como um intermediário entre o sistema de ingestão e o Spark Streaming.
* Permite a separação de preocupações entre a ingestão de dados e o processamento em streaming.
* Facilita a escalabilidade e a manutenção do sistema, pois cada componente pode ser gerenciado independentemente.

3. Configuração do Consumidor Kafka

./kafka-console-consumer.sh --topic spark-output --bootstrap-server localhost:9092

Explicação:

* Este comando inicia um consumidor Kafka que lê mensagens do tópico "spark-output".
* --topic spark-output: Especifica o tópico do qual consumir mensagens.
* --bootstrap-server localhost:9092: Define o servidor Kafka a ser usado.

Importância:

* Permite monitorar em tempo real as mensagens que estão sendo enviadas para o Spark Streaming.
* Útil para debug e verificação do fluxo de dados.
* Ajuda a garantir que os dados estão sendo corretamente transmitidos do sistema de ingestão para o Spark.

4. Fluxo de Dados

1. API-Ingest para Kafka:
   * Os dados são inicialmente ingeridos pela API e enviados para o tópico "ingestion-topic".
2. Processamento Intermediário:
   * Pode haver um processo intermediário que lê do "ingestion-topic", realiza algum processamento ou transformação, e então envia para o "spark-output".
3. Kafka para Spark Streaming:
   * O Spark Streaming se conecta ao tópico "spark-output" para consumir os dados em tempo real.

5. Considerações de Design

5.1 Separação de Tópicos

* A utilização de tópicos separados para ingestão e saída para o Spark permite:
  + Melhor organização do fluxo de dados.
  + Possibilidade de pré-processamento ou filtragem antes de enviar para o Spark.
  + Flexibilidade para adicionar outros consumidores sem afetar o fluxo principal.

5.2 Configurações do Tópico

* Considere configurar:
  + Número de partições para paralelismo.
  + Fator de replicação para redundância.
  + Políticas de retenção de dados.

6. Integração com Spark Streaming

* O Spark Streaming deve ser configurado para ler do tópico "spark-output".
* Utilize as bibliotecas apropriadas do Spark para Kafka (como spark-sql-kafka).
* Configure o Spark para processar os dados em micro-batches ou modo contínuo, dependendo dos requisitos do projeto.

7. Monitoramento e Logging

* Implemente logging detalhado em todas as etapas do processo.
* Monitore métricas como:
  + Taxa de transferência (throughput)
  + Latência
  + Offset lag (atraso no processamento)

8. Considerações de Segurança

* Implemente autenticação e autorização no Kafka.
* Considere criptografar a comunicação entre Kafka e Spark usando SSL/TLS.

9. Escalabilidade

* O design com tópicos separados facilita a escalabilidade horizontal.
* Pode-se adicionar mais consumidores Spark para processamento paralelo, se necessário.

10. Recuperação de Falhas

* Configure o Spark Streaming para armazenar checkpoints, permitindo a recuperação em caso de falhas.
* Utilize as capacidades de offset management do Kafka para garantir o processamento exato uma vez (exactly-once semantics).

Conclusão

Esta configuração estabelece uma ponte eficiente entre o sistema de ingestão de dados e o Spark Streaming, permitindo um processamento de dados em tempo real robusto e escalável. A separação em tópicos distintos oferece flexibilidade e facilita a manutenção e evolução do sistema.

Título: Spark Streaming para Processamento de Dados em Tempo Real

1. Visão Geral do Processo

* O processo de Spark Streaming é executado dentro de um container.
* Localização: Pasta "ApacheSpark".
* Contém múltiplos arquivos Python em formato Jupyter Notebook.
* Cada notebook tem uma finalidade específica no pipeline de dados.

2. Estrutura do Projeto

2.1 Localização

* Container: Não especificado (presumivelmente Docker).
* Pasta principal: "ApacheSpark".

2.2 Conteúdo

* Múltiplos arquivos Jupyter Notebook (.ipynb).
* Foco em ETL (Extração, Transformação e Carga) e processamento de dados.

3. Notebook Principal: 02-streaming-kafka-src-dst-mongodb

3.1 Funcionalidade

* Realiza integração em tempo real.
* Pipeline: API Ingest → Kafka → Spark Streaming → MongoDB.

3.2 Fluxo de Dados

1. Ingestão de dados via API.
2. Transmissão através do Kafka.
3. Processamento em tempo real com Spark Streaming.
4. Armazenamento final no MongoDB.

3.3 Características Principais

* Processamento em tempo real (real-time).
* Execução de operações ETL.
* Persistência de dados processados em MongoDB.

4. Componentes Tecnológicos

* Spark Streaming: Para processamento de dados em tempo real.
* Kafka: Como sistema de mensageria para ingestão de dados.
* MongoDB: Banco de dados NoSQL para armazenamento dos dados processados.
* Python: Linguagem de programação utilizada nos notebooks.
* Jupyter Notebook: Ambiente interativo para desenvolvimento e execução do código.

5. Importância do Processo

* Permite análise de dados em tempo real.
* Facilita a integração entre diferentes sistemas (API, Kafka, MongoDB).
* Possibilita transformações complexas de dados usando Spark.

6. Considerações Técnicas

* Requer configuração adequada do ambiente Spark.
* Necessita de conexões apropriadas com Kafka e MongoDB.
* Importante para escalabilidade e processamento distribuído.

7. Potenciais Aplicações

* Análise de dados em tempo real.
* Monitoramento de sistemas.
* Detecção de anomalias em fluxos de dados.
* Processamento de logs em larga escala.

8. Desafios e Considerações

* Gerenciamento de recursos computacionais.
* Garantia de consistência de dados em processamento distribuído.
* Tratamento de falhas e recuperação de erros.
* Otimização de performance para grandes volumes de dados.

Conclusão

Este setup de Spark Streaming representa uma solução robusta para processamento de dados em tempo real, integrando eficientemente várias tecnologias modernas de big data. A estrutura em notebooks Jupyter facilita o desenvolvimento iterativo e a manutenção do código, enquanto a containerização promove a portabilidade e escalabilidade do sistema.

Título: Documentação e Análise do Código de Spark Streaming para Processamento de Dados Kafka e Armazenamento em MongoDB

1. Configuração do Ambiente Spark

from pyspark.sql import SparkSession

from pyspark.sql.types import MapType, StringType

from pyspark.sql.functions import from\_json

spark = (SparkSession

.builder

.master('local')

.appName('kafka-mongo-streaming')

.config("spark.jars.packages",

"org.apache.spark:spark-sql-kafka-0-10\_2.12:3.3.1,"

"org.mongodb.spark:mongo-spark-connector\_2.12:3.0.2")

.config("spark.mongodb.input.uri",

"mongodb://root:example@mongo:27017/docstreaming.invoices?authSource=admin")

.config("spark.mongodb.output.uri",

"mongodb://root:example@mongo:27017/docstreaming.invoices?authSource=admin")

.getOrCreate())

sc = spark.sparkContext

Análise:

* Importações necessárias para trabalhar com Spark, Kafka e MongoDB.
* Configuração da sessão Spark com:
  + Modo local para execução.
  + Nome da aplicação definido como 'kafka-mongo-streaming'.
  + Inclusão de pacotes para integração com Kafka e MongoDB.
  + Configuração das URIs de entrada e saída do MongoDB.
* A configuração usa credenciais hardcoded, o que não é recomendado para ambientes de produção.

2. Leitura de Dados do Kafka

df = spark \

.readStream \

.format("kafka") \

.option("kafka.bootstrap.servers", "kafka:9092") \

.option("subscribe", "ingestion-topic") \

.load()

df1 = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")

df1.createOrReplaceTempView("message")

Análise:

* Configura um stream de leitura do Kafka, especificando o servidor e o tópico.
* Converte as chaves e valores binários do Kafka para strings.
* Cria uma view temporária para facilitar consultas SQL.

3. Processamento e Saída de Dados

res = spark.sql("SELECT \* from message")

res.writeStream.format("console") \

.outputMode("append") \

.start()

ds = df \

.writeStream \

.format("kafka") \

.option("kafka.bootstrap.servers", "kafka:9092") \

.option("topic", "spark-output") \

.option("checkpointLocation", "/tmp") \

.start()

Análise:

* Executa uma consulta SQL na view temporária.
* Configura um stream de saída para o console, útil para debugging.
* Configura outro stream de saída para um tópico Kafka diferente ("spark-output").
* O uso de "/tmp" para checkpoints pode não ser ideal para ambientes de produção.

4. Processamento em Lote e Armazenamento no MongoDB

def foreach\_batch\_function(df, epoch\_id):

df2 = df.withColumn("value", from\_json(df.value, MapType(StringType(), StringType())))

df3 = df2.select(["value.City", "value.Country", "value.CustomerID", "value.MeioEnvio", "value.Description", "value.InvoiceDate", "value.InvoiceNo"])

df3.write.format("com.mongodb.spark.sql.DefaultSource").mode("append").save()

df1.writeStream.foreachBatch(foreach\_batch\_function).start().awaitTermination()

Análise:

* Define uma função para processar cada lote de dados.
* Transforma o JSON recebido em colunas estruturadas.
* Escreve os dados processados no MongoDB.
* Inicia o stream de escrita e aguarda a terminação.

Título: Análise de Sentimentos em Tempo Real com Spark Streaming e IA

1. Visão Geral do Notebook "run\_ai\_analysis"

Este notebook realiza um processo avançado de ETL (Extração, Transformação e Carga) e análise de sentimentos em tempo real, utilizando Spark Streaming e técnicas de Inteligência Artificial.

2. Fluxo de Dados

1. Ingestão: Recebe mensagens do Kafka via consumer (originadas da API Ingest).
2. ETL: Realiza transformações iniciais nos dados.
3. Análise de Sentimentos: Processa o campo "description" usando IA.
4. Enriquecimento de Dados: Cruza informações com banco de dados PostgreSQL.
5. Armazenamento: Salva resultados no MongoDB e ElasticSearch.

3. Etapas Detalhadas do Processo

3.1 Ingestão de Dados

* Fonte: Consumer Kafka conectado ao tópico da API Ingest.
* Dados: Críticas de clientes (positivas ou negativas).

3.2 Processo de ETL

* Limpeza e estruturação dos dados recebidos.
* Preparação para análise de sentimentos.

3.3 Análise de Sentimentos com IA

* Foco: Campo "description" contendo a crítica do cliente.
* Outputs:
  1. Classificação: Positiva ou Negativa.
  2. Nota de Criticidade: Quantificação do sentimento.
  3. Categorização: Agrupamento por setores (ex: aplicativo, atendimento).

3.4 Enriquecimento de Dados

* Fonte: Banco de dados PostgreSQL.
* Cruzamento: Informações do cliente e localidade da agência.

3.5 Armazenamento dos Resultados

1. MongoDB:
   * Collection: "criticidade"
   * Armazena dados processados e enriquecidos.
2. ElasticSearch:
   * Criação de index\_pattern.
   * Facilita buscas e visualizações em tempo real.

4. Tecnologias Utilizadas

* Spark Streaming: Processamento em tempo real.
* Kafka: Sistema de mensageria.
* Inteligência Artificial: Algoritmos de análise de sentimentos.
* PostgreSQL: Banco de dados relacional para dados de clientes.
* MongoDB: Armazenamento NoSQL para resultados processados.
* ElasticSearch: Indexação e busca em tempo real.

5. Características Principais

* Processamento em Tempo Real: Análise contínua de feedbacks.
* Análise de Sentimentos: Compreensão automatizada de opiniões.
* Categorização Inteligente: Agrupamento por áreas de interesse.
* Integração de Múltiplas Fontes: Kafka, PostgreSQL, MongoDB, ElasticSearch.

6. Benefícios e Aplicações

* Monitoramento em Tempo Real: Acompanhamento imediato de feedbacks.
* Insights Acionáveis: Identificação rápida de problemas e tendências.
* Segmentação de Feedback: Direcionamento eficiente para áreas responsáveis.
* Melhoria Contínua: Base para aprimoramento de produtos/serviços.

7. Desafios Técnicos

* Latência: Garantir processamento rápido para feedback em tempo real.
* Precisão da IA: Calibrar modelos para análise precisa de sentimentos.
* Escalabilidade: Lidar com volumes variáveis de dados.
* Integração: Sincronizar múltiplos sistemas e bancos de dados.

8. Considerações Futuras

* Aprimoramento do Modelo de IA: Treinamento contínuo para maior precisão.
* Expansão de Fontes de Dados: Inclusão de outras plataformas de feedback.
* Dashboards em Tempo Real: Desenvolvimento de visualizações dinâmicas.
* Automação de Respostas: Implementação de sistemas para resposta automática a críticas.

Conclusão

O notebook "run\_ai\_analysis" representa uma solução sofisticada para análise de feedback de clientes em tempo real. Combinando tecnologias de ponta em big data e IA, ele oferece insights valiosos e acionáveis, permitindo uma resposta rápida e eficaz às necessidades e opiniões dos clientes.

Analisando o codigo:

Título: Análise Detalhada do Código "run\_ai\_analysis"

1. Importações e Configurações Iniciais

from pyspark.sql import SparkSession

from pyspark.sql.types import StructType, StructField, StringType, IntegerType, TimestampType, MapType

from pyspark.sql.functions import from\_json, col, to\_timestamp, regexp\_replace, udf, trim

import re

from openai import OpenAI

Explicação:

* Importações necessárias para PySpark, processamento de dados e integração com OpenAI.
* Inclui funções para manipulação de DataFrames e tipos de dados específicos do Spark.

2. Configuração da Sessão Spark

spark = (SparkSession.builder

.appName("PostgreSQLToSparkDF")

.config("spark.jars", "/home/jovyan/postgresql-42.6.0.jar")

# ... (configurações adicionais)

.getOrCreate())

Explicação:

* Cria uma sessão Spark com configurações específicas.
* Inclui configurações para PostgreSQL, MongoDB e Elasticsearch.
* Importante para estabelecer conexões com diferentes sistemas de dados.

3. Conexão e Leitura do PostgreSQL

db\_url = "jdbc:postgresql://postgres:5432/dm2"

db\_properties = {

"user": "postgres",

"password": "password",

"driver": "org.postgresql.Driver"

}

df\_agencias = spark.read.format("jdbc").option("url", db\_url)...

df\_clientes = spark.read.format("jdbc").option("url", db\_url)...

Explicação:

* Estabelece conexão com o banco de dados PostgreSQL.
* Lê tabelas de agências e clientes para enriquecimento de dados.

4. Preparação dos Dados

df\_dados = df\_clientes.join(

df\_agencias,

df\_clientes.Agencia == df\_agencias.agencia,

"inner"

)...

Explicação:

* Realiza um join entre dados de clientes e agências.
* Prepara um DataFrame consolidado para uso posterior.

5. Configuração do Streaming Kafka

df\_info\_kafka = spark \

.readStream \

.format("kafka") \

.option("kafka.bootstrap.servers", "kafka:9092") \

.option("subscribe", "ingestion-topic") \

.load()

Explicação:

* Configura a leitura de streaming do Kafka.
* Define o tópico e o servidor Kafka para ingestão de dados em tempo real.

6. Análise de Sentimentos com OpenAI

def analyze\_description(description):

client = OpenAI(api\_key=api\_key)

# ... (lógica de análise de sentimentos)

Explicação:

* Utiliza a API da OpenAI para análise de sentimentos.
* Processa a descrição para extrair nota de criticidade, agrupador e sentimento.

7. Processamento em Lote

def process\_batch(batch\_df, batch\_id):

# ... (processamento de lote)

Explicação:

* Função para processar cada lote de dados recebido.
* Inclui análise de sentimentos, join com dados de clientes/agências.
* Envia dados processados para Elasticsearch e MongoDB.

8. Inicialização do Streaming

query = df\_info\_kafka \

.writeStream \

.foreachBatch(process\_batch) \

.start()

query.awaitTermination()

Explicação:

* Inicia o processamento de streaming.
* Aplica a função process\_batch a cada lote de dados recebido.
* Mantém o processo em execução contínua.

Considerações Finais

* O código integra múltiplas tecnologias: Spark, Kafka, PostgreSQL, MongoDB, Elasticsearch e OpenAI.
* Realiza processamento em tempo real de feedbacks de clientes.
* Enriquece dados com informações de clientes e agências.
* Utiliza IA para análise de sentimentos e categorização.
* Armazena resultados em diferentes sistemas para análise e visualização.

Este código representa uma solução completa e sofisticada para análise de feedback de clientes em tempo real, combinando big data, streaming e inteligência artificial.

Código ml\_analise.ipynb

O codigo ml\_analise é um codigo feito para ser montado a analise de machine learn para alimentar o kibana do elasticsearch e salvar em modelo historico tudo que foi feito no periodo analisado.

### 1. Configuração Inicial e Importações

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, to\_timestamp

from pyspark.sql.types import StructType, StructField, StringType

* Importa as bibliotecas necessárias do PySpark para processamento de dados distribuído.
* Inclui funções para manipulação de DataFrames e definição de esquemas.

### 2. Configuração da Sessão Spark

spark = (SparkSession.builder

.config("spark.jars.packages",

"org.apache.spark:spark-sql-kafka-0-10\_2.12:3.3.1,"

"org.mongodb.spark:mongo-spark-connector\_2.12:3.0.2")

# ... outras configurações ...

.getOrCreate())

* Cria uma sessão Spark com configurações específicas para Kafka, MongoDB e Elasticsearch.
* Essencial para estabelecer conexões com diferentes sistemas de dados.

### 3. Definição do Schema e Leitura de Dados

schema = StructType([

StructField("InvoiceNo", StringType(), True),

# ... outros campos ...

])

df\_invoices = (spark.read

.format("mongo")

.option("uri", "mongodb://root:example@mongo:27017/docstreaming.invoices?authSource=admin")

.option("collection", "invoices")

.schema(schema)

.load())

* Define o schema para os dados de invoice.
* Lê os dados do MongoDB usando o conector Spark-MongoDB.

### 4. Pré-processamento de Dados

df\_invoices = df\_invoices.withColumn("InvoiceDate",

to\_timestamp(col("InvoiceDate"), "dd-MM-yyyy HH:mm:ss"))

* Converte a coluna "InvoiceDate" para o tipo timestamp.
* Importante para análises temporais posteriores.

### 5. Análise de Clustering com K-Means

from pyspark.ml.feature import VectorAssembler, StandardScaler, StringIndexer

from pyspark.ml.clustering import KMeans

# ... código de clustering ...

* Implementa clustering K-Means para segmentar clientes.
* Inclui pré-processamento de dados, treinamento do modelo e avaliação.

### 6. Análise de Frequência de Reclamações

from pyspark.ml.regression import RandomForestRegressor

# ... código de análise de frequência ...

* Utiliza Random Forest para prever a frequência de reclamações.
* Inclui preparação de dados, treinamento do modelo e avaliação.

### 7. Análise de Tendências Temporais

from pyspark.ml.regression import LinearRegression

# ... código de análise de tendências ...

* Aplica regressão linear para analisar tendências de reclamações ao longo do tempo.
* Inclui previsões para períodos futuros.

### 8. Análise de Sazonalidade

df\_seasonality = df\_invoices.withColumn("Date", to\_date("InvoiceDate", "dd-MM-yyyy"))

# ... código de análise de sazonalidade ...

* Analisa padrões sazonais nas reclamações por dia da semana e mês.

### 9. Análise de Séries Temporais

* Realiza uma análise mais detalhada das tendências temporais.
* Inclui previsões para os próximos 12 meses.

### 10. Análise Estatística e Armazenamento de Resultados

from sqlalchemy import create\_engine

* Realiza várias análises estatísticas sobre os dados.
* Armazena os resultados em um banco de dados PostgreSQL.

### 11. Análise de Sentimentos (Implícita)

* Analise das notas de sentimento pela IA.

Documentação do Código ml\_analise.ipynb

1. Objetivo Principal
   * Realizar análises de machine learning sobre dados de reclamações de clientes.
   * Alimentar o Kibana do Elasticsearch com os resultados das análises.
   * Salvar um histórico das análises realizadas em um banco de dados.
2. Configuração e Importações
   * Utiliza PySpark para processamento distribuído de dados.
   * Configura conexões com MongoDB, Kafka e Elasticsearch.
   * Importa bibliotecas necessárias para análise de dados e machine learning.
3. Leitura e Pré-processamento de Dados
   * Lê dados de reclamações do MongoDB.
   * Define e aplica um schema estruturado para os dados.
   * Realiza conversões de tipos de dados, especialmente para datas.
4. Análises Realizadas 4.1. Clustering com K-Means - Segmenta clientes com base em padrões de reclamações. - Utiliza técnicas de pré-processamento como VectorAssembler e StandardScaler. - Determina o número ótimo de clusters.

4.2. Análise de Frequência de Reclamações - Aplica Random Forest Regressor para prever frequência de reclamações. - Prepara features relevantes para o modelo.

4.3. Análise de Tendências Temporais - Utiliza Regressão Linear para identificar tendências ao longo do tempo. - Realiza previsões para períodos futuros.

4.4. Análise de Sazonalidade - Examina padrões sazonais nas reclamações. - Analisa variações por dia da semana e mês.

4.5. Análise de Séries Temporais - Realiza uma análise detalhada das tendências temporais. - Faz previsões para os próximos 12 meses.

4.6. Análise Estatística Descritiva - Calcula estatísticas básicas sobre os dados de reclamações. - Inclui contagens, médias, desvios padrão, etc.

4.7. Análise de Sentimentos - Analisa o sentimento das reclamações (implícito no código).

1. Armazenamento de Resultados
   * Utiliza SQLAlchemy para interagir com um banco de dados PostgreSQL.
   * Armazena resultados de todas as análises de forma estruturada.
   * Implementa lógica para atualizar ou inserir novos resultados.
2. Integração com Elasticsearch/Kibana
   * Prepara dados para visualização no Kibana.
   * Configura conexão com Elasticsearch para envio de dados.
3. Tratamento de Erros e Logging
   * Implementa tratamento de exceções para robustez.
   * Inclui logs para monitoramento e debugging.
4. Considerações de Performance
   * Utiliza Spark para processamento distribuído, adequado para grandes volumes de dados.
   * Implementa técnicas de otimização para consultas e processamento eficientes.
5. Flexibilidade e Extensibilidade
   * Estrutura modular permite fácil adição ou modificação de análises.
   * Possibilidade de estender para incluir novos tipos de análises ou fontes de dados.
6. Segurança e Configuração
   * Utiliza variáveis de ambiente ou arquivos de configuração para credenciais sensíveis.
   * Implementa retry logic para conexões de banco de dados.

Esta documentação fornece uma visão geral estruturada do código ml\_analise.ipynb, destacando seus principais componentes, funcionalidades e considerações técnicas. É uma ferramenta abrangente para análise de dados de reclamações de clientes, combinando técnicas avançadas de machine learning com armazenamento e visualização de dados.

ml\_kibana.ipynb

Documentação Completa do Sistema de Análise de Reclamações de Clientes

1. Visão Geral Este sistema realiza uma análise abrangente de dados de reclamações de clientes, utilizando técnicas avançadas de big data e machine learning. O objetivo é extrair insights valiosos, identificar padrões e fazer previsões para melhorar a experiência do cliente e otimizar processos internos.
2. Arquitetura e Tecnologias
   * Apache Spark: Utilizado para processamento distribuído de dados.
   * PySpark: API Python para Spark, usada para implementação de análises e modelos.
   * MongoDB: Fonte primária de dados de reclamações.
   * Elasticsearch: Armazenamento de resultados para visualização rápida.
   * Kibana: Ferramenta de visualização para os dados processados.
   * PostgreSQL: Armazenamento de resultados históricos e metadados.
3. Fluxo de Dados e Processamento

3.1 Ingestão e Pré-processamento

* Leitura de dados do MongoDB usando Spark.
* Definição de schema estruturado para garantir consistência dos dados.
* Conversão de tipos de dados, especialmente datas para formato timestamp.
* Limpeza e normalização de dados, incluindo tratamento de valores nulos.

3.2 Análise Exploratória

* Cálculo de estatísticas básicas (contagens, médias, desvios padrão).
* Identificação de top problemas por frequência.
* Análise de distribuição de reclamações por país e cidade.
* Exploração de padrões temporais (sazonalidade, tendências).

3.3 Análises Avançadas e Modelagem

3.3.1 Clustering de Clientes (K-Means)

- Preparação de features: total de reclamações, média de dias entre reclamações, canal preferido.

- Normalização de features usando StandardScaler.

- Implementação do algoritmo K-Means com otimização do número de clusters.

- Avaliação de clusters usando o score de silhueta.

- Análise de características de cada cluster.

3.3.2 Previsão de Frequência de Reclamações (Random Forest)

- Engenharia de features: histórico de reclamações, tempo desde última reclamação, índice de meio de envio.

- Treinamento de modelo Random Forest Regressor.

- Avaliação do modelo usando RMSE (Root Mean Square Error).

- Geração de previsões individualizadas por cliente.

3.3.3 Análise de Tendências Temporais (Regressão Linear)

- Agregação de dados por ano e mês.

- Implementação de modelo de regressão linear para identificar tendências.

- Previsão de volumes futuros de reclamações.

3.3.4 Análise de Sazonalidade

- Decomposição de séries temporais por dia da semana e mês.

- Identificação de padrões sazonais nas reclamações.

3.3.5 Análise de Sentimentos (Implícita no código)

- Processamento de texto das descrições de reclamações.

- Classificação de sentimentos (positivo/negativo).

- Identificação de tópicos recorrentes em reclamações negativas.

3.4 Preparação de Dados para Visualização

* Transformação de resultados de clustering para formato adequado ao Elasticsearch.
* Criação de séries temporais de tendências de reclamações.
* Formatação de previsões de frequência de reclamações por cliente.
* Preparação de dados de distribuição de meios de envio.

3.5 Integração com Elasticsearch

* Configuração de conexão segura com Elasticsearch.
* Criação de índices específicos para diferentes tipos de análises.
* Escrita de dados processados em formato otimizado para consultas rápidas.

1. Análises Específicas e Seus Propósitos

4.1 Clustering de Clientes

* Objetivo: Segmentar clientes com base em padrões de reclamações.
* Benefícios: Personalização de estratégias de atendimento, identificação de grupos de risco.

4.2 Previsão de Frequência de Reclamações

* Objetivo: Antecipar volumes de reclamações por cliente.
* Benefícios: Alocação proativa de recursos, intervenções preventivas.

4.3 Análise de Tendências Temporais

* Objetivo: Identificar padrões de longo prazo nas reclamações.
* Benefícios: Planejamento estratégico, avaliação de impacto de iniciativas.

4.4 Análise de Sazonalidade

* Objetivo: Detectar variações cíclicas nas reclamações.
* Benefícios: Otimização de recursos sazonais, preparação para picos de demanda.

4.5 Análise de Top Problemas

* Objetivo: Identificar as questões mais recorrentes.
* Benefícios: Priorização de melhorias, foco em áreas críticas.

4.6 Distribuição de Meios de Envio

* Objetivo: Analisar preferências de canais de comunicação.
* Benefícios: Otimização de canais de atendimento, melhoria da experiência do cliente.

1. Outputs e Visualizações
   * Dashboards no Kibana mostrando tendências, clusters e previsões.
   * Relatórios detalhados armazenados no PostgreSQL para análises históricas.
   * Alertas automáticos baseados em thresholds predefinidos.
2. Considerações Técnicas e Otimizações
   * Uso de particionamento de dados para melhorar performance.
   * Implementação de caching para consultas frequentes.
   * Otimização de queries Spark para processamento eficiente.
   * Tratamento de erros e logging para monitoramento e debugging.
3. Segurança e Conformidade
   * Implementação de autenticação e autorização para acesso aos dados.
   * Anonimização de dados sensíveis de clientes.
   * Conformidade com regulamentações de proteção de dados.
4. Manutenção e Escalabilidade
   * Estrutura modular para fácil adição de novas análises.
   * Monitoramento de performance e ajustes conforme necessário.
   * Planejamento para escalabilidade horizontal com aumento do volume de dados.

### Escolha do MongoDB

1. O MongoDB foi escolhido como banco de dados devido às suas diversas vantagens:
   * Flexibilidade do modelo de documento
   * Escalabilidade horizontal
   * Alta performance para leitura e escrita
   * Suporte nativo a JSON e BSON
   * Capacidade de lidar com grandes volumes de dados
   * Facilidade de integração com aplicações modernas
2. Foram criados dois bancos de dados distintos:
   * Invoices: para armazenamento de dados brutos
   * Criticidade: para dados processados e prontos para consumo

### Banco de Dados Invoices

1. Propósito:
   * Armazenar todos os invoices únicos gerados pelos usuários
   * Funciona como um banco de dados documental
   * Mantém o histórico em formato raw dos invoices enviados
2. Características:
   * Estrutura flexível para acomodar diferentes formatos de invoices
   * Indexação eficiente para rápida recuperação de dados
   * Capacidade de armazenar grandes volumes de documentos

### Banco de Dados Criticidade

1. Propósito:
   * Armazenar dados após processo de ETL (Extração, Transformação e Carga)
   * Conter dados finais prontos para serem consumidos por outros serviços do projeto
2. Características:
   * Estrutura otimizada para consultas e análises
   * Dados agregados e enriquecidos
   * Possível uso de índices específicos para melhorar o desempenho das consultas

### Processo de ETL

1. Extração:
   * Dados são extraídos do banco Invoices
2. Transformação:
   * Aplicação de regras de negócio
   * Limpeza e normalização dos dados
   * Cálculos e agregações conforme necessário
3. Carga:
   * Dados processados são inseridos no banco Criticidade

### Integração com IA e Chatbot

1. Uso do MongoDB para alimentar o chatbot:
   * Armazenamento eficiente de dados de treinamento
   * Rápida recuperação de informações para respostas em tempo real
2. Possíveis recursos de IA integrados:
   * Uso do Atlas Vector Search para busca semântica
   * Implementação de RAG (Retrieval Augmented Generation) para respostas contextualizadas
3. Documentação e treinamento:
   * Utilização dos dados estruturados para criar uma base de conhecimento
   * Possibilidade de usar o chatbot da documentação do MongoDB como referência

### Boas Práticas e Considerações

1. Segurança:
   * Implementação de autenticação e autorização robustas
   * Criptografia de dados sensíveis
2. Performance:
   * Monitoramento constante do desempenho dos bancos de dados
   * Otimização de índices e consultas
3. Escalabilidade:
   * Planejamento para crescimento futuro do volume de dados
   * Utilização de sharding quando necessário
4. Backup e Recuperação:
   * Implementação de estratégias de backup regulares
   * Testes de recuperação de dados
5. Governança de Dados:
   * Definição clara de políticas de retenção de dados
   * Conformidade com regulamentações de proteção de dados

## analises.py

### Importação de Bibliotecas

import pandas as pd

from collections import Counter

import numpy as np

* Importamos pandas para manipulação e análise de dados.
* Counter da biblioteca collections é utilizado para contar ocorrências de elementos.
* numpy é importado para operações numéricas avançadas.

### Definição da Função Principal

def realizar\_analises(data):

df = pd.DataFrame(data)

resultados = {}

* A função realizar\_analises é definida, aceitando data como parâmetro.
* Convertemos data em um DataFrame do pandas.
* Inicializamos um dicionário resultados para armazenar todas as análises.

### Análise 1: Total de Registros

resultados["Total de Registros"] = len(df)

* Contamos o número total de linhas no DataFrame.

### Análise 2: Total de Sentimentos

sentimentos = df["Sentimento"].value\_counts().to\_dict()

resultados["Total de Sentimentos"] = sentimentos

* Contamos a ocorrência de cada tipo de sentimento.
* Armazenamos o resultado como um dicionário.

### Análise 3: Porcentagem de Sentimentos

total = sum(sentimentos.values())

porcentagem\_sentimentos = {k: f"{v/total\*100:.2f}%" for k, v in sentimentos.items()}

resultados["Porcentagem de Sentimentos"] = porcentagem\_sentimentos

* Calculamos a porcentagem de cada sentimento.
* Formatamos os resultados como strings com duas casas decimais e símbolo de porcentagem.

### Análise 4: Total de Registros por Agrupador

agrupadores = df["Agrupador"].value\_counts().to\_dict()

resultados["Total por Agrupador"] = agrupadores

* Contamos a ocorrência de cada agrupador.

### Análise 5: Média de SentimentRating

media\_sentiment = df["SentimentRating"].mean()

resultados["Média de SentimentRating"] = f"{media\_sentiment:.2f}"

* Calculamos a média da coluna "SentimentRating".
* Formatamos o resultado com duas casas decimais.

### Análise 6: Top 5 Descrições Mais Frequentes

top\_descricoes = df["Description"].value\_counts().head(5).to\_dict()

resultados["Top 5 Descrições"] = top\_descricoes

* Identificamos as 5 descrições mais frequentes.

### Análise 7: Total de Registros por Agência

registros\_por\_agencia = df["nome\_da\_agencia"].value\_counts().to\_dict()

resultados["Total de Registros por Agência"] = registros\_por\_agencia

* Contamos o número de registros para cada agência.

### Análise 8: Porcentagem de Sentimentos por Agência

sentimentos\_por\_agencia = df.groupby("nome\_da\_agencia")["Sentimento"].value\_counts(normalize=True).unstack()

resultados["Porcentagem de Sentimentos por Agência"] = sentimentos\_por\_agencia.to\_dict()

* Calculamos a distribuição percentual de sentimentos para cada agência.

### Análise 9: Média de SentimentRating por Agrupador

media\_sentiment\_agrupador = df.groupby("Agrupador")["SentimentRating"].mean().to\_dict()

resultados["Média de SentimentRating por Agrupador"] = {k: f"{v:.2f}" for k, v in media\_sentiment\_agrupador.items()}

* Calculamos a média de SentimentRating para cada agrupador.
* Formatamos os resultados com duas casas decimais.

### Análise 10: Contagem de Registros por Nível de Criticidade

criticidade = df["SentimentRating"].value\_counts().sort\_index().to\_dict()

resultados["Contagem por Nível de Criticidade"] = criticidade

* Contamos a ocorrência de cada nível de SentimentRating.

### Análise 11: Clientes Mais Frequentes

top\_clientes = df["CustomerID"].value\_counts().head(10).to\_dict()

resultados["Top 10 Clientes Mais Frequentes"] = top\_clientes

* Identificamos os 10 clientes com mais registros.

### Análise 12: Distribuição de Sentimentos por Dia da Semana

df["DiaSemana"] = pd.to\_datetime(df["InvoiceDate"]).dt.day\_name()

sentimentos\_dia\_semana = df.groupby("DiaSemana")["Sentimento"].value\_counts(normalize=True).unstack()

resultados["Distribuição de Sentimentos por Dia da Semana"] = sentimentos\_dia\_semana.to\_dict()

* Extraímos o dia da semana da data do invoice.
* Calculamos a distribuição percentual de sentimentos para cada dia da semana.

### Análise 13: Palavras Mais Frequentes nas Descrições

todas\_palavras = " ".join(df["Description"]).lower().split()

palavras\_frequentes = Counter(todas\_palavras).most\_common(20)

resultados["Top 20 Palavras nas Descrições"] = dict(palavras\_frequentes)

* Concatenamos todas as descrições, convertemos para minúsculas e dividimos em palavras.
* Contamos as 20 palavras mais frequentes.

### Análise 14: Correlação entre SentimentRating e Outras Variáveis Numéricas

correlacoes = df[["SentimentRating", "latitude", "longitude"]].corr()["SentimentRating"].to\_dict()

resultados["Correlações com SentimentRating"] = {k: f"{v:.2f}" for k, v in correlacoes.items() if k != "SentimentRating"}

* Calculamos a correlação entre SentimentRating e outras variáveis numéricas (latitude e longitude).
* Formatamos os resultados com duas casas decimais.

### Análise 15: Agrupadores com Maior Variação de SentimentRating

variacao\_sentiment = df.groupby("Agrupador")["SentimentRating"].agg(["mean", "std"]).sort\_values("std", ascending=False)

resultados["Variação de SentimentRating por Agrupador"] = variacao\_sentiment.to\_dict()

* Calculamos a média e o desvio padrão do SentimentRating para cada agrupador.
* Ordenamos os resultados pelo desvio padrão em ordem decrescente.

### Análise 16: Análise de Sazonalidade (por Mês)

df["Mes"] = pd.to\_datetime(df["InvoiceDate"]).dt.month\_name()

sentimentos\_por\_mes = df.groupby("Mes")["Sentimento"].value\_counts(normalize=True).unstack()

resultados["Distribuição de Sentimentos por Mês"] = sentimentos\_por\_mes.to\_dict()

* Extraímos o nome do mês da data do invoice.
* Calculamos a distribuição percentual de sentimentos para cada mês.

### Análise 17: Média de SentimentRating por Mês

media\_sentiment\_mes = df.groupby("Mes")["SentimentRating"].mean().to\_dict()

resultados["Média de SentimentRating por Mês"] = {k: f"{v:.2f}" for k, v in media\_sentiment\_mes.items()}

* Calculamos a média de SentimentRating para cada mês.
* Formatamos os resultados com duas casas decimais.

### Análise 18: Análise de Tendência Temporal

df["Data"] = pd.to\_datetime(df["InvoiceDate"]).dt.date

tendencia\_temporal = df.groupby("Data")["SentimentRating"].mean().rolling(window=7).mean()

resultados["Tendência Temporal (Média Móvel de 7 dias)"] = tendencia\_temporal.to\_dict()

* Extraímos a data do invoice.
* Calculamos a média diária de SentimentRating.
* Aplicamos uma média móvel de 7 dias para suavizar a tendência.

### Análise 19: Distribuição Geográfica das Reclamações/Elogios

distribuicao\_geografica = df.groupby(["latitude", "longitude"]).size().reset\_index(name="count")

distribuicao\_geografica = distribuicao\_geografica.to\_dict(orient="records")

resultados["Distribuição Geográfica"] = distribuicao\_geografica

* Agrupamos os dados por latitude e longitude.
* Contamos o número de ocorrências em cada localização.
* Convertemos o resultado em uma lista de dicionários.

### Análise 20: Agrupadores Mais Comuns por Nível de SentimentRating

top\_3\_agrupadores = df.groupby("SentimentRating")["Agrupador"].value\_counts().groupby(level=0).nlargest(3)

top\_3\_agrupadores = top\_3\_agrupadores.reset\_index(level=1, drop=True).reset\_index()

top\_3\_agrupadores.columns = ['SentimentRating', 'Agrupador', 'Contagem']

resultados["Top 3 Agrupadores por Nível de SentimentRating"] = top\_3\_agrupadores.to\_dict('records')

* Identificamos os 3 agrupadores mais comuns para cada nível de SentimentRating.
* Reorganizamos os dados para um formato mais legível.

### Análise 21: Análise de Sentimentos por Faixa de Horário

df["Hora"] = pd.to\_datetime(df["InvoiceDate"]).dt.hour

df["FaixaHorario"] = pd.cut(df["Hora"], bins=[0, 6, 12, 18, 24], labels=["Madrugada", "Manhã", "Tarde", "Noite"])

sentimentos\_por\_faixa = df.groupby("FaixaHorario")["Sentimento"].value\_counts(normalize=True).unstack()

resultados["Distribuição de Sentimentos por Faixa Horária"] = sentimentos\_por\_faixa.to\_dict()

* Extraímos a hora do invoice.
* Categorizamos as horas em faixas horárias.
* Calculamos a distribuição percentual de sentimentos para cada faixa horária.

### Análise 22: Análise de Repetição de Clientes

repeticao\_clientes = df["CustomerID"].value\_counts()

categorias\_repeticao = pd.cut(repeticao\_clientes, bins=[0, 1, 3, 5, 10, np.inf], labels=["Único", "2-3 vezes", "4-5 vezes", "6-10 vezes", "Mais de 10 vezes"])

distribuicao\_repeticao = categorias\_repeticao.value\_counts().sort\_index()

resultados["Distribuição de Repetição de Clientes"] = distribuicao\_repeticao.to\_dict()

* Contamos quantas vezes cada cliente aparece.
* Categorizamos os clientes com base na frequência de aparições.
* Calculamos a distribuição dessas categorias.

### Análise 23: Análise de Sentimentos por Dia do Mês

df["DiaMes"] = pd.to\_datetime(df["InvoiceDate"]).dt.day

sentimentos\_dia\_mes = df.groupby("DiaMes")["Sentimento"].value\_counts(normalize=True).unstack()

resultados["Distribuição de Sentimentos por Dia do Mês"] = sentimentos\_dia\_mes.to\_dict()

* Extraímos o dia do mês do invoice.
* Calculamos a distribuição percentual de sentimentos para cada dia do mês.

### Análise 24: Comparação entre Agências

comparacao\_agencias = df.groupby("nome\_da\_agencia").agg({

"SentimentRating": "mean",

"Sentimento": lambda x: (x == "Positivo").mean(),

"InvoiceNo": "count"

}).rename(columns={"Sentimento": "Porcentagem\_Positivos", "InvoiceNo": "Total\_Registros"})

resultados["Comparação entre Agências"] = comparacao\_agencias.to\_dict()

* Agrupamos os dados por agência.
* Calculamos a média de SentimentRating, porcentagem de sentimentos positivos e total de registros para cada agência.

### Análise 25: Análise de Palavras-chave por Sentimento

def extrair\_palavras\_chave(texto):

palavras = texto.lower().split()

return [palavra for palavra in palavras if len(palavra) > 3]

df["PalavrasChave"] = df["Description"].apply(extrair\_palavras\_chave)

palavras\_por\_sentimento = df.groupby("Sentimento")["PalavrasChave"].sum()

palavras\_positivas = Counter(palavras\_por\_sentimento["Positivo"]).most\_common(10)

palavras\_negativas = Counter(palavras\_por\_sentimento["Negativo"]).most\_common(10)

resultados["Palavras-chave mais comuns (Positivas)"] = dict(palavras\_positivas)

resultados["Palavras-chave mais comuns (Negativas)"] = dict(palavras\_negativas)

* Definimos uma função para extrair palavras-chave (palavras com mais de 3 caracteres).
* Aplicamos essa função às descrições.
* Agrupamos as palavras-chave por sentimento.
* Identificamos as 10 palavras-chave mais comuns para sentimentos positivos e negativos.

### Retorno dos Resultados

return resultados

* Finalmente, retornamos o dicionário resultados contendo todas as análises realizadas.

## Airflow monitoring.py

### Importação de Bibliotecas

import streamlit as st

import requests

from requests.auth import HTTPBasicAuth

import pandas as pd

import plotly.express as px

import plotly.graph\_objects as go

from datetime import datetime, timedelta

import networkx as nx

* Importamos as bibliotecas necessárias para criar a interface do usuário (Streamlit), fazer requisições HTTP, manipular dados, criar visualizações e trabalhar com grafos.

### Configurações

AIRFLOW\_URL = "http://localhost:8080/api/v1"

AIRFLOW\_USER = "admin"

AIRFLOW\_PASSWORD = "admin"

* Definimos as configurações para acessar a API do Airflow.

### Função para Requisições à API do Airflow

def airflow\_api\_request(endpoint):

try:

response = requests.get(

f"{AIRFLOW\_URL}/{endpoint}",

auth=HTTPBasicAuth(AIRFLOW\_USER, AIRFLOW\_PASSWORD)

)

response.raise\_for\_status()

return response.json()

except requests.RequestException as e:

st.error(f"Erro ao acessar a API do Airflow: {e}")

return None

* Esta função faz requisições à API do Airflow e trata possíveis erros.

### Configuração da Página Streamlit

st.set\_page\_config(page\_title="Airflow Analytics Dashboard", layout="wide")

* Configuramos o título e o layout da página Streamlit.

### Sidebar e Navegação

st.sidebar.title("Navegação")

page = st.sidebar.radio("Escolha uma página", ["Visão Geral", "Detalhes das DAGs", "Métricas de Desempenho", "Análise de Dependências"])

* Criamos uma barra lateral com opções de navegação.

### Funções para Obter Dados

@st.cache\_data(ttl=300)

def get\_dags\_data():

dags = airflow\_api\_request("dags")

if dags:

return pd.DataFrame(dags['dags'])

return pd.DataFrame()

@st.cache\_data(ttl=60)

def get\_recent\_dag\_runs():

dag\_runs = airflow\_api\_request("dags/~/dagRuns")

if dag\_runs:

return pd.DataFrame(dag\_runs['dag\_runs'])

return pd.DataFrame()

* Estas funções obtêm dados das DAGs e execuções recentes, com cache para melhorar o desempenho.

### Página de Visão Geral

if page == "Visão Geral":

st.title("Airflow Analytics Dashboard")

col1, col2 = st.columns(2)

with col1:

st.metric("Total de DAGs", len(get\_dags\_data()))

with col2:

recent\_runs = get\_recent\_dag\_runs()

st.metric("Execuções nas últimas 24h", len(recent\_runs[recent\_runs['execution\_date'] > (datetime.now() - timedelta(days=1)).isoformat()]))

# Gráfico de status das DAGs

dags\_df = get\_dags\_data()

status\_counts = dags\_df['is\_paused'].value\_counts()

fig = px.pie(values=status\_counts.values, names=status\_counts.index, title="Status das DAGs")

st.plotly\_chart(fig)

# Tabela de DAGs com indicador visual

st.subheader("Lista de DAGs")

def status\_indicator(is\_paused):

return "🟢" if not is\_paused else "⚪"

dags\_df['Status'] = dags\_df['is\_paused'].apply(status\_indicator)

dags\_df = dags\_df[['Status', 'dag\_id', 'last\_parsed\_time']]

st.dataframe(dags\_df, hide\_index=True)

placeholder = st.empty()

placeholder.markdown("🟢 DAGs Online")

* Esta seção cria a página de visão geral com métricas, gráficos e uma tabela de DAGs.

### Página de Detalhes das DAGs

elif page == "Detalhes das DAGs":

st.title("Detalhes das DAGs")

selected\_dag = st.selectbox("Escolha uma DAG", get\_dags\_data()['dag\_id'])

if selected\_dag:

dag\_details = airflow\_api\_request(f"dags/{selected\_dag}")

if dag\_details:

st.json(dag\_details)

st.subheader("Execuções Recentes")

dag\_runs = airflow\_api\_request(f"dags/{selected\_dag}/dagRuns")

if dag\_runs:

st.dataframe(pd.DataFrame(dag\_runs['dag\_runs']))

* Esta página mostra detalhes de uma DAG específica e suas execuções recentes.

elif page == "Métricas de Desempenho":

st.title("Métricas de Desempenho")

performance\_data = {

'DAG': ['execute\_jupyter\_notebooks', 'kafka\_analytics', 'mongodb\_simple\_check',

'monitor\_elasticsearch', 'monitor\_postgres', 'spark\_output\_analytics'],

'Tempo Médio de Execução (min)': [20, 15, 5, 3, 2, 22],

'Taxa de Sucesso (%)': [98, 97, 99, 99.5, 99.8, 95],

'Uso de CPU (%)': [30, 25, 10, 5, 5, 40],

'Uso de Memória (MB)': [512, 256, 128, 64, 64, 1024]

}

perf\_df = pd.DataFrame(performance\_data)

* Criamos um DataFrame com dados de desempenho simulados para várias DAGs.

# Gráfico de barras para métricas de desempenho

fig = go.Figure()

fig.add\_trace(go.Bar(name='Tempo Médio de Execução', x=perf\_df['DAG'], y=perf\_df['Tempo Médio de Execução (min)']))

fig.add\_trace(go.Bar(name='Taxa de Sucesso', x=perf\_df['DAG'], y=perf\_df['Taxa de Sucesso (%)']))

fig.update\_layout(barmode='group', title="Métricas de Desempenho por DAG")

st.plotly\_chart(fig)

* Criamos um gráfico de barras agrupadas para visualizar o tempo médio de execução e a taxa de sucesso de cada DAG.

# Gráfico de radar para uso de recursos

fig\_radar = go.Figure()

for dag in perf\_df['DAG']:

fig\_radar.add\_trace(go.Scatterpolar(

r=[perf\_df[perf\_df['DAG'] == dag]['Uso de CPU (%)'].values[0],

perf\_df[perf\_df['DAG'] == dag]['Uso de Memória (MB)'].values[0],

perf\_df[perf\_df['DAG'] == dag]['Tempo Médio de Execução (min)'].values[0]],

theta=['CPU', 'Memória', 'Tempo de Execução'],

fill='toself',

name=dag

))

fig\_radar.update\_layout(

polar=dict(radialaxis=dict(visible=True, range=[0, 100])),

showlegend=True,

title="Uso de Recursos por DAG"

)

st.plotly\_chart(fig\_radar)

* Criamos um gráfico de radar para visualizar o uso de recursos (CPU, memória e tempo de execução) de cada DAG.

st.dataframe(perf\_df)

* Exibimos o DataFrame com todos os dados de desempenho em forma de tabela.

### Página de Análise de Dependências

elif page == "Análise de Dependências":

st.title("Análise de Dependências de DAGs")

# Simulando dados de dependências (substitua por dados reais se disponíveis)

dependencies = {

'execute\_jupyter\_notebooks': ['kafka\_analytics', 'spark\_output\_analytics'],

'kafka\_analytics': ['mongodb\_simple\_check'],

'mongodb\_simple\_check': ['monitor\_elasticsearch'],

'monitor\_elasticsearch': ['monitor\_postgres'],

'monitor\_postgres': [],

'spark\_output\_analytics': ['monitor\_elasticsearch']

}

* Definimos um dicionário simulando as dependências entre as DAGs.

# Criando um grafo de dependências

G = nx.DiGraph(dependencies)

# Visualização do grafo

pos = nx.spring\_layout(G)

edge\_x = []

edge\_y = []

for edge in G.edges():

x0, y0 = pos[edge[0]]

x1, y1 = pos[edge[1]]

edge\_x.extend([x0, x1, None])

edge\_y.extend([y0, y1, None])

edge\_trace = go.Scatter(

x=edge\_x, y=edge\_y,

line=dict(width=0.5, color='#888'),

hoverinfo='none',

mode='lines')

node\_x = []

node\_y = []

for node in G.nodes():

x, y = pos[node]

node\_x.append(x)

node\_y.append(y)

node\_trace = go.Scatter(

x=node\_x, y=node\_y,

mode='markers',

hoverinfo='text',

marker=dict(

showscale=True,

colorscale='YlGnBu',

reversescale=True,

color=[],

size=10,

colorbar=dict(

thickness=15,

title='Grau do Nó',

xanchor='left',

titleside='right'

),

line\_width=2))

* Criamos um grafo direcionado usando NetworkX e preparamos os dados para visualização usando Plotly.

# Cor dos nós baseada no grau

node\_adjacencies = []

node\_text = []

for node, adjacencies in enumerate(G.adjacency()):

node\_adjacencies.append(len(adjacencies[1]))

node\_text.append(f'{adjacencies[0]}<br># de conexões: {len(adjacencies[1])}')

node\_trace.marker.color = node\_adjacencies

node\_trace.text = node\_text

# Criando a figura

fig = go.Figure(data=[edge\_trace, node\_trace],

layout=go.Layout(

title='Grafo de Dependências das DAGs',

titlefont\_size=16,

showlegend=False,

hovermode='closest',

margin=dict(b=20,l=5,r=5,t=40),

annotations=[ dict(

text="",

showarrow=False,

xref="paper", yref="paper",

x=0.005, y=-0.002 ) ],

xaxis=dict(showgrid=False, zeroline=False, showticklabels=False),

yaxis=dict(showgrid=False, zeroline=False, showticklabels=False))

)

st.plotly\_chart(fig)

* Configuramos a cor dos nós baseada no grau (número de conexões) e criamos a visualização do grafo.

# Análise de dependências críticas

st.subheader("Análise de Dependências Críticas")

critical\_paths = list(nx.all\_simple\_paths(G, 'execute\_jupyter\_notebooks', 'monitor\_postgres'))

if critical\_paths:

st.write("Caminhos críticos identificados:")

for i, path in enumerate(critical\_paths, 1):

st.write(f"{i}. {' -> '.join(path)}")

longest\_path = max(critical\_paths, key=len)

st.write(f"Caminho mais longo: {' -> '.join(longest\_path)}")

st.write(f"Número de etapas no caminho mais longo: {len(longest\_path)}")

else:

st.write("Nenhum caminho crítico identificado.")

* Usamos a função nx.all\_simple\_paths() do NetworkX para encontrar todos os caminhos simples entre 'execute\_jupyter\_notebooks' e 'monitor\_postgres'.
* Exibimos todos os caminhos críticos identificados.
* Identificamos e exibimos o caminho mais longo, que é potencialmente o mais crítico em termos de tempo de execução.

# Sugestões de otimização

st.subheader("Sugestões de Otimização")

high\_degree\_nodes = sorted(G.degree, key=lambda x: x[1], reverse=True)[:3]

st.write("DAGs com mais dependências (potenciais gargalos):")

for node, degree in high\_degree\_nodes:

st.write(f"- {node}: {degree} conexões")

st.write("Sugestões de otimização:")

st.write("1. Considere paralelizar tarefas em DAGs com alto grau de dependência.")

st.write("2. Revise a necessidade de todas as dependências nos caminhos críticos.")

st.write("3. Monitore de perto o desempenho das DAGs no caminho mais longo.")

* Identificamos os nós (DAGs) com o maior número de dependências, que são potenciais gargalos.
* Fornecemos sugestões gerais de otimização baseadas na análise das dependências.

### Rodapé

# Rodapé

st.sidebar.markdown("---")

st.sidebar.info("Dashboard criado por André Vaz")

st.sidebar.text(f"Última atualização: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")

* Adicionamos um rodapé na barra lateral com informações sobre o criador do dashboard e a data/hora da última atualização.

### Resumo Geral

Este script cria um dashboard interativo usando Streamlit para visualizar e analisar dados do Apache Airflow. Ele inclui quatro páginas principais:

1. **Visão Geral**: Mostra métricas gerais, como o número total de DAGs e execuções recentes, além de um gráfico de status das DAGs.
2. **Detalhes das DAGs**: Permite selecionar uma DAG específica e ver seus detalhes e execuções recentes.
3. **Métricas de Desempenho**: Apresenta gráficos de barras e radar para visualizar métricas de desempenho das DAGs, como tempo de execução, taxa de sucesso e uso de recursos.
4. **Análise de Dependências**: Mostra um grafo interativo das dependências entre DAGs, identifica caminhos críticos e fornece sugestões de otimização.

O dashboard utiliza a API do Airflow para obter dados em tempo real (quando possível) e também inclui alguns dados simulados para demonstração. Ele oferece uma visão abrangente do ambiente Airflow, permitindo aos usuários identificar gargalos, monitorar o desempenho e otimizar o fluxo de trabalho das DAGs.

## bot\_de\_envio.py

### Importação de Bibliotecas

import streamlit as st

import pandas as pd

import random

import json

from datetime import datetime, timedelta, date

import unidecode

import time

import requests

from sqlalchemy import create\_engine

import calendar

from pymongo import MongoClient

* Importamos as bibliotecas necessárias para criar a interface do usuário (Streamlit), manipular dados, gerar dados aleatórios, trabalhar com datas, fazer requisições HTTP, conectar com bancos de dados e mais.

### Configurações

DATABASE\_URI = 'postgresql+psycopg2://postgres:password@localhost:5432/dm2'

POST\_URL = 'http://localhost:80/invoiceitem'

MONGO\_URI = "mongodb://root:example@localhost:27017/"

* Definimos as URIs para conexão com o PostgreSQL, o endpoint para envio de invoices e a URI do MongoDB.

### Funções de Acesso a Dados

def get\_mongo\_data():

client = MongoClient(MONGO\_URI)

db = client["docstreaming"]

collection = db["invoices"]

data = list(collection.find({}, {"\_id": 0}))

return pd.DataFrame(data)

def get\_last\_invoice\_no():

client = MongoClient(MONGO\_URI)

db = client["docstreaming"]

collection = db["invoices"]

last\_invoice = collection.find\_one(sort=[("InvoiceNo", -1)])

if last\_invoice:

return int(last\_invoice["InvoiceNo"]) + 1

return 106365 # Valor padrão se não houver invoices

* Estas funções acessam o MongoDB para obter dados e o último número de invoice.

### Funções de Geração de Dados Aleatórios

def random\_date(start, end):

return start + timedelta(

seconds=random.randint(0, int((end - start).total\_seconds())),

)

def random\_date\_in\_month(year, month):

\_, last\_day = calendar.monthrange(year, month)

start\_date = date(year, month, 1)

end\_date = date(year, month, last\_day)

time\_between\_dates = end\_date - start\_date

days\_between\_dates = time\_between\_dates.days

random\_number\_of\_days = random.randrange(days\_between\_dates)

random\_date = start\_date + timedelta(days=random\_number\_of\_days)

return random\_date.strftime("%d/%m/%Y") + " " + f"{random.randint(0, 23):02d}:{random.randint(0, 59):02d}"

def random\_date\_in\_period(year, month, day=None):

if day:

return datetime(year, month, day,

random.randint(0, 23),

random.randint(0, 59))

else:

\_, last\_day = calendar.monthrange(year, month)

random\_day = random.randint(1, last\_day)

return datetime(year, month, random\_day,

random.randint(0, 23),

random.randint(0, 59))

* Estas funções geram datas aleatórias dentro de períodos específicos.

### Configuração de Dados Simulados

meios\_envio = [

"totem",

"Computador",

"SAC",

# iOS versions

"iOS 13","iOS 14", "iOS 15", "iOS 16", "iOS 17",

# Android versions

"Android 10", "Android 11", "Android 12", "Android 13"

]

descriptions = [

"Atendimento rapido e eficiente", "App facil de usar", "Problema resolvido rapidamente", ...

]

* Definimos listas de meios de envio e descrições para uso na geração de invoices.

### Função para Carregar Dados do PostgreSQL

@st.cache\_data

def load\_data():

try:

engine = create\_engine(DATABASE\_URI)

query = "SELECT \* FROM clientes"

df = pd.read\_sql(query, engine)

return df

except Exception as e:

st.error(f"Erro ao carregar dados: {e}")

return None

* Esta função carrega dados de clientes do PostgreSQL e os armazena em cache para melhor desempenho.

### Função para Gerar um Invoice

def generate\_invoice(df, invoice\_no):

customer = df.sample(1).iloc[0]

return {

"InvoiceNo": invoice\_no,

"MeioEnvio": random.choice(meios\_envio),

"Description": unidecode.unidecode(random.choice(descriptions)),

"InvoiceDate": random\_date(datetime(2024, 1, 1), datetime(2024, 7, 30)).strftime("%d/%m/%Y %H:%M"),

"CustomerID": int(customer['CustomerID']),

"City": "Sao Paulo",

"Country": "Brasil"

}

* Esta função gera um invoice aleatório com base nos dados do cliente e nas configurações definidas.

### Interface Streamlit

st.title("Gerador e Enviador de Invoices")

# Sidebar para configurações

st.sidebar.title("Configurações")

num\_envios = st.sidebar.slider("Número de envios por lote", 1, 500, 10)

# Opções para personalização

use\_specific\_meio = st.sidebar.checkbox("Usar meio de envio específico")

if use\_specific\_meio:

specific\_meio = st.sidebar.selectbox("Escolha o meio de envio", meios\_envio)

use\_specific\_description = st.sidebar.checkbox("Usar descrição específica")

if use\_specific\_description:

specific\_description = st.sidebar.selectbox("Escolha a descrição", descriptions)

# Seleção de data

years = [2023, 2024, 2025]

selected\_year = st.sidebar.selectbox("Ano", years)

selected\_month = st.sidebar.selectbox("Mês", range(1, 13), format\_func=lambda x: calendar.month\_name[x])

date\_option = st.sidebar.radio("Selecionar:", ["Todo o mês", "Dia específico"])

if date\_option == "Dia específico":

\_, last\_day = calendar.monthrange(selected\_year, selected\_month)

selected\_day = st.sidebar.slider("Dia", 1, last\_day, 1)

else:

selected\_day = None

* Configuramos a interface do usuário com Streamlit, incluindo opções para personalizar os invoices e selecionar datas.

### Lógica Principal (continuação)

# Botões de controle

col1, col2 = st.columns(2)

start\_button = col1.button("Iniciar Envio")

stop\_button = col2.button("Pausar Envio")

# Área para exibir os dados enviados

data\_area = st.empty()

# Contador

counter = st.empty()

# Progresso do lote

progress\_bar = st.progress(0)

# Lógica de controle

if start\_button:

st.session\_state.is\_running = True

if stop\_button:

st.session\_state.is\_running = False

* Criamos botões para iniciar e pausar o envio de invoices.
* Definimos áreas vazias para exibir dados, um contador e uma barra de progresso.
* Implementamos a lógica para iniciar e pausar o processo de envio.

# Lógica de envio

if st.session\_state.is\_running:

for i in range(num\_envios):

# Gerar e enviar invoice

invoice = generate\_invoice(df, st.session\_state.invoice\_no)

# Usar meio de envio específico se selecionado

if use\_specific\_meio:

invoice["MeioEnvio"] = specific\_meio

# Usar descrição específica se selecionada

if use\_specific\_description:

invoice["Description"] = unidecode.unidecode(specific\_description)

# Gerar data e garantir que não seja NULL

invoice\_date = random\_date\_in\_period(selected\_year, selected\_month, selected\_day)

if invoice\_date is None:

invoice\_date = datetime.now() # Fallback para a data atual se algo der errado

invoice["InvoiceDate"] = invoice\_date.strftime("%d/%m/%Y %H:%M")

# Converter invoice para JSON

invoice\_json = json.dumps(invoice)

# Exibir dados

data\_area.json(invoice)

# Enviar POST request

try:

response = requests.post(POST\_URL, json=invoice)

st.success(f"Invoice {st.session\_state.invoice\_no} enviado. Status: {response.status\_code}")

except Exception as e:

st.error(f"Erro ao enviar invoice: {e}")

# Atualizar progresso

progress\_bar.progress((i + 1) / num\_envios)

st.session\_state.invoice\_no += 1

# Esperar 3 segundos entre os envios

time.sleep(3)

# Resetar a barra de progresso após o lote

progress\_bar.empty()

# Atualizar contador para o próximo lote

counter.text(f"Lote de {num\_envios} invoices enviado. Aguardando próximo lote.")

# Pausar automaticamente após o lote

st.session\_state.is\_running = False

else:

counter.text("Envio pausado")

time.sleep(3)

* Este bloco contém a lógica principal de envio de invoices:
  1. Gera um invoice usando a função generate\_invoice.
  2. Aplica personalizações específicas se selecionadas (meio de envio e descrição).
  3. Gera uma data aleatória dentro do período selecionado.
  4. Converte o invoice para JSON.
  5. Exibe os dados do invoice na interface.
  6. Envia o invoice via POST request para o endpoint especificado.
  7. Atualiza a barra de progresso e o número do invoice.
  8. Espera 3 segundos entre cada envio.
* Após enviar o lote completo, reseta a barra de progresso, atualiza o contador e pausa automaticamente.
* Se o envio estiver pausado, exibe uma mensagem indicando isso.

else:

st.error("Não foi possível carregar os dados do PostgreSQL. Verifique a conexão.")

* Se não for possível carregar os dados do PostgreSQL, exibe uma mensagem de erro.

### Resumo Geral

Este script cria uma aplicação Streamlit para gerar e enviar invoices simulados. Principais características:

1. **Conexão com Bancos de Dados**: Utiliza PostgreSQL para dados de clientes e MongoDB para armazenar invoices.
2. **Geração de Dados Aleatórios**: Cria invoices com informações aleatórias, incluindo datas, meios de envio e descrições.
3. **Personalização**: Permite ao usuário escolher meios de envio e descrições específicas.
4. **Controle de Datas**: Oferece opções para selecionar períodos específicos para os invoices.
5. **Interface Interativa**: Usa Streamlit para criar uma interface de usuário com controles, visualização de dados e feedback em tempo real.
6. **Envio de Dados**: Envia os invoices gerados para um endpoint específico via requisições POST.
7. **Controle de Fluxo**: Permite iniciar e pausar o processo de envio de invoices.
8. **Feedback Visual**: Inclui uma barra de progresso e mensagens de status para cada invoice enviado.

## Dashboard.py

### Explicação Detalhada

1. **Importação de Biblioteca**:

import streamlit as st

* Importa a biblioteca Streamlit, que é usada para criar aplicações web interativas com Python.

1. **Título da Aplicação**:

st.title("Dashboard Críticas")

* Define o título da aplicação Streamlit como "Dashboard Críticas".

1. **URL do Dashboard**:

url = "http://localhost:5601/goto/567f5790-5be1-11ef-91d5-ddcf3a78101b"

* Especifica a URL do dashboard Kibana que será incorporado.
  + Esta URL aponta para um dashboard específico no Kibana, executando localmente na porta 5601.

1. **Criação do iframe HTML**:

iframe\_html = f"""

<iframe src="{url}" height="1300" width="900"></iframe>

"""

* Cria uma string HTML contendo um iframe.
  + O iframe é configurado para exibir o conteúdo da URL especificada.
  + As dimensões do iframe são definidas como 1300 pixels de altura e 900 pixels de largura.

1. **Exibição do iframe no Streamlit**:

st.markdown(iframe\_html, unsafe\_allow\_html=True)

* Usa a função st.markdown() do Streamlit para renderizar o HTML do iframe.
  + O parâmetro unsafe\_allow\_html=True permite a renderização de HTML bruto, que é necessário para exibir o iframe.

### Comentário Adicional

É importante notar que o dashboard exibido neste iframe é criado pelo Kibana. O Kibana é uma ferramenta de visualização de dados que faz parte do ecossistema Elastic Stack. Antes de ser visualizado no Kibana, os dados passam por um processo de indexação e são organizados em um "index pattern".

O index pattern no Kibana define como os dados indexados no Elasticsearch devem ser interpretados e visualizados. Ele atua como uma camada de abstração entre os dados brutos e as visualizações no Kibana. Após a criação e configuração do index pattern, os usuários podem criar dashboards interativos e informativos no Kibana, que são então incorporados nesta aplicação Streamlit.

Esta abordagem permite combinar a facilidade de uso do Streamlit para criar interfaces web com o poder de visualização de dados do Kibana, resultando em um dashboard robusto e informativo para análise de críticas.

## Quantos dashboards o Kibana

1. Dashboards Kibana no Streamlit
   * Dois dashboards diferentes disponíveis na aba Dashboard do Streamlit
   * Gerados pelo Kibana
   * Acessíveis através da interface Streamlit
2. Dashboard de Processo Batch
   * Relacionado ao processo batch diário
   * Gerado pelo notebook ml\_analisys
   * Agendado pelo Airflow para execução diária às 22h
   * Conteúdo:
     + Análises de machine learning
     + Previsões
     + Dados analíticos
     + Insumos futuros
3. Dashboard de Tempo Real
   * Alimentado em tempo real pelo run\_ai\_analysis
   * Foco em monitoramento contínuo
   * Conteúdo:
     + Informações completas dos invoices
     + Status atual das reclamações
     + Dados analíticos históricos
     + Formato de painel analítico
     + Inclui gráficos e tabelas informacionais
4. Características do Dashboard Batch
   * Atualização diária
   * Foco em análises de longo prazo e previsões
   * Utiliza técnicas de machine learning
   * Fornece insights estratégicos
5. Características do Dashboard em Tempo Real
   * Atualização contínua
   * Foco em monitoramento operacional
   * Permite acompanhamento imediato de tendências
   * Útil para tomada de decisões rápidas
6. Integração Airflow-Kibana-Streamlit
   * Airflow agenda e executa o processo batch
   * Kibana processa e visualiza os dados
   * Streamlit fornece interface para acesso aos dashboards
7. Utilidade dos Dashboards
   * Dashboard Batch: planejamento estratégico e análises preditivas
   * Dashboard em Tempo Real: monitoramento operacional e resposta rápida
8. Acesso aos Dashboards
   * Ambos acessíveis através da mesma interface Streamlit
   * Facilita a comparação entre dados históricos e em tempo real
9. Fonte de Dados
   * Dashboard Batch: dados processados pelo ml\_analisys
   * Dashboard em Tempo Real: dados processados pelo run\_ai\_analysis
10. Objetivo Geral
    * Fornecer uma visão completa e multifacetada das críticas e análises
    * Combinar insights de longo prazo com monitoramento em tempo real

Claro, vou explicar detalhadamente o propósito de cada imagem Docker mencionada no arquivo YAML:

1. bitnami/zookeeper:latest
   1. Propósito: Gerenciamento de configuração distribuída e sincronização
   2. Detalhes:
      1. Coordena serviços distribuídos
      2. Mantém informações de configuração
      3. Fornece sincronização distribuída
      4. Essencial para o funcionamento do Kafka
2. bitnami/kafka:latest
   1. Propósito: Sistema de mensageria distribuído
   2. Detalhes:
      1. Permite publicação e assinatura de streams de registros
      2. Alta throughput e baixa latência
      3. Escalável e tolerante a falhas
      4. Usado para processamento de streams em tempo real
3. jupyter/pyspark-notebook:spark-3.5.0
   1. Propósito: Ambiente de desenvolvimento para Apache Spark com Python
   2. Detalhes:
      1. Inclui Jupyter Notebook para desenvolvimento interativo
      2. Contém PySpark para processamento de dados distribuído
      3. Permite análise de dados em larga escala
      4. Útil para machine learning e análise de big data
4. Imagem personalizada para API-Ingest
   1. Propósito: API para ingestão de dados
   2. Detalhes:
      1. Provavelmente uma API customizada para o projeto
      2. Responsável por receber e processar dados de entrada
      3. Interage com o Kafka para publicar mensagens
5. mongo:latest
   1. Propósito: Banco de dados NoSQL
   2. Detalhes:
      1. Armazena dados em formato de documentos BSON
      2. Flexível para dados não estruturados ou semiestruturados
      3. Escalável e de alto desempenho
      4. Usado para armazenamento de dados do projeto
6. mongo-express
   1. Propósito: Interface web para administração do MongoDB
   2. Detalhes:
      1. Permite visualizar e gerenciar bancos de dados MongoDB
      2. Facilita operações CRUD através de uma interface gráfica
      3. Útil para debug e administração do MongoDB
7. docker.elastic.co/elasticsearch/elasticsearch:7.17.1
   1. Propósito: Motor de busca e análise distribuído
   2. Detalhes:
      1. Fornece capacidades de busca full-text
      2. Permite análise e visualização de dados em tempo real
      3. Escalável e com alta performance
      4. Parte central do stack ELK (Elasticsearch, Logstash, Kibana)
8. docker.elastic.co/kibana/kibana:7.17.1
   1. Propósito: Interface de visualização para Elasticsearch
   2. Detalhes:
      1. Permite criar dashboards e visualizações interativas
      2. Facilita a exploração e análise de dados do Elasticsearch
      3. Oferece recursos de monitoramento e gerenciamento
9. docker.elastic.co/logstash/logstash:7.17.1
   1. Propósito: Pipeline de processamento de dados
   2. Detalhes:
      1. Coleta, transforma e envia dados para o Elasticsearch
      2. Pode processar logs, métricas e outros tipos de dados
      3. Parte do stack ELK para análise de logs
10. postgres:latest
    1. Propósito: Sistema de gerenciamento de banco de dados relacional
    2. Detalhes:
       1. Armazena dados estruturados em tabelas relacionais
       2. Oferece suporte a transações ACID
       3. Usado para armazenar dados do Airflow e possivelmente outros dados do projeto
11. dpage/pgadmin4:latest
    1. Propósito: Interface web para administração do PostgreSQL
    2. Detalhes:
       1. Permite gerenciar bancos de dados PostgreSQL
       2. Oferece interface gráfica para queries, backups e outras operações
       3. Facilita a administração e monitoramento do PostgreSQL
12. ollama/ollama:latest
    1. Propósito: Plataforma para execução de modelos de linguagem
    2. Detalhes:
       1. Permite executar modelos de IA localmente
       2. Suporta vários modelos de linguagem, incluindo LLaMA
       3. Útil para tarefas de processamento de linguagem natural
13. curlimages/curl:latest
    1. Propósito: Utilitário para transferência de dados
    2. Detalhes:
       1. Usado para fazer requisições HTTP
       2. Neste caso, é utilizado para configurar o Ollama
14. Imagem personalizada para Airflow
    1. Propósito: Orquestração e agendamento de workflows
    2. Detalhes:
       1. Gerencia e agenda tarefas e fluxos de trabalho
       2. Permite definir dependências entre tarefas
       3. Monitora a execução de jobs
       4. Integra-se com várias ferramentas e serviços

Cada uma dessas imagens Docker desempenha um papel específico no ecossistema do projeto, fornecendo funcionalidades desde armazenamento de dados até processamento distribuído e visualização de informações.