3<https://opposite-rudbeckia-e0f.notion.site/Financial-Transactions-Anomaly-Detection-584e4c1d89664e79bb3f8f1ff87a725d?pvs=4>

import pandas as pd

import numpy as np

# Step 1: Data Preprocessing

def load\_and\_preprocess\_data(file\_path):

# Load the dataset

df = pd.read\_csv(file\_path)

# Handling missing values by removing rows with NaN values

df.dropna(inplace=True)

# Standardize date format if there's a date column (e.g., 'date')

if 'date' in df.columns:

df['date'] = pd.to\_datetime(df['date'], errors='coerce')

# Drop rows with invalid dates

df.dropna(subset=['date'], inplace=True)

return df

# Step 2: Statistical Analysis

def calculate\_statistics(df, group\_by\_column='category'):

# Group by the specified column and calculate statistics

grouped = df.groupby(group\_by\_column)['amount']

stats = grouped.agg(['mean', 'median', 'std']).reset\_index()

return stats

def establish\_thresholds(df, method='z\_score', threshold=3):

if method == 'z\_score':

df['z\_score'] = (df['amount'] - df['amount'].mean()) / df['amount'].std()

df['anomaly'] = np.abs(df['z\_score']) > threshold

elif method == 'iqr':

Q1 = df['amount'].quantile(0.25)

Q3 = df['amount'].quantile(0.75)

IQR = Q3 - Q1

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

df['anomaly'] = (df['amount'] < lower\_bound) | (df['amount'] > upper\_bound)

return df

# Step 3: Anomaly Detection

def detect\_anomalies(df, method='z\_score', threshold=3):

# Establish thresholds for detecting anomalies

df = establish\_thresholds(df, method, threshold)

# Filter anomalies

anomalies = df[df['anomaly']]

return anomalies

# Main function

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

file\_path = 'C:/Users/HP/Desktop/financial\_transactions.csv'

df = load\_and\_preprocess\_data(file\_path)

stats = calculate\_statistics(df)

print("Statistics:\n", stats)

anomalies = detect\_anomalies(df)

print("Anomalies:\n", anomalies[['transaction\_id', 'date', 'category', 'amount']])

# Save anomalies to a CSV file

anomalies.to\_csv('anomalies.csv', index=False)

output

Statistics:

category mean median std

0 Entertainment 200.0 200.0 NaN

1 Food 1015.0 25.0 1719.062244

2 Transport 45.0 45.0 NaN

3 Utilities 142.5 142.5 10.606602

Anomalies:

Empty DataFrame

Columns: [transaction\_id, date, category, amount]

Index:

Csv file path= 'C:/Users/HP/Desktop/financial\_transactions.csv

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| transaction\_id |  | date |  | category | amount |  |
| TRX001 |  | 24-06-2001 |  | Food | 25 |  |
| TRX002 |  | 24-06-2001 |  | Utilities | 150 |  |
| TRX003 |  | 24-06-2001 |  | Entertainment | 200 |  |
| TRX004 |  | 24-06-2002 |  | Food | 3000 |  |
| TRX005 |  | 24-06-2002 |  | Transport | 45 |  |
| TRX006 |  | 24-06-2003 |  | Utilities | 135 |  |
| TRX007 |  | 24-06-2001 |  | Food | 20 |  |

Question 2

.<https://opposite-rudbeckia-e0f.notion.site/Port-Sail-Calculation-6e1bc5859b8548418f37f38071a28bf6?pvs=4>

Extract data for the specified vessel and voyage, and exclude records with non-null allocatedVoyageId

SELECT \* FROM voyages

WHERE

 allocatedVoyageId = 2 OR allocatedVoyageId IS NULL

Calculate precise UTC date-times for events and generate time durations between key events.

import sqlite3

from datetime import datetime, timedelta

# Connect to the SQLite database (or any other DB you are using)

conn = sqlite3.connect('voyages.db')

cursor = conn.cursor()

# Fetch the data from the table

cursor.execute("SELECT id, event, dateStamp, timeStamp FROM voyages")

rows = cursor.fetchall()

# Function to convert dateStamp and timeStamp to a precise UTC datetime

def convert\_to\_utc(dateStamp, timeStamp):

base\_date = datetime(1899, 12, 30) # Excel's epoch date

event\_date = base\_date + timedelta(days=dateStamp + timeStamp)

return event\_date

# Calculate UTC date-times for events

events = {}

for row in rows:

event\_id = row[0]

event\_name = row[1]

dateStamp = row[2]

timeStamp = row[3]

utc\_datetime = convert\_to\_utc(dateStamp, timeStamp)

events[event\_id] = (event\_name, utc\_datetime)

# Calculate durations between key events (SOSP to EOSP for same voyage\_Id)

for event\_id, (event\_name, event\_time) in events.items():

if event\_name == 'SOSP':

for other\_event\_id, (other\_event\_name, other\_event\_time) in events.items():

if other\_event\_name == 'EOSP' and event\_id < other\_event\_id:

duration = other\_event\_time - event\_time

print(f"Duration between {event\_name} at {event\_time} and {other\_event\_name} at {other\_event\_time}: {duration}")

# Close the database connection

conn.close()

Identify and segment different voyage stages based on a series of 'SOSP' (Start of Sea Passage) and 'EOSP' (End of Sea Passage) events.

import sqlite3

from datetime import datetime, timedelta

# Connect to the SQLite database

conn = sqlite3.connect('voyages.db')

cursor = conn.cursor()

# Fetch the data from the table

cursor.execute("SELECT id, event, dateStamp, timeStamp, voyage\_Id FROM voyages ORDER BY dateStamp, timeStamp")

rows = cursor.fetchall()

# Function to convert dateStamp and timeStamp to a precise UTC datetime

def convert\_to\_utc(dateStamp, timeStamp):

base\_date = datetime(1899, 12, 30) # Excel's epoch date

event\_date = base\_date + timedelta(days=dateStamp + timeStamp)

return event\_date

# Identify and segment different voyage stages

voyages = {}

for row in rows:

event\_id, event\_name, dateStamp, timeStamp, voyage\_Id = row

utc\_datetime = convert\_to\_utc(dateStamp, timeStamp)

if voyage\_Id not in voyages:

voyages[voyage\_Id] = []

voyages[voyage\_Id].append((event\_name, utc\_datetime))

# Calculate durations between key events for each voyage

for voyage\_Id, events in voyages.items():

start\_time = None

for event\_name, event\_time in events:

if event\_name == 'SOSP':

start\_time = event\_time

elif event\_name == 'EOSP' and start\_time is not None:

duration = event\_time - start\_time

print(f"Voyage {voyage\_Id}: Duration between SOSP at {start\_time} and EOSP at {event\_time}: {duration}")

start\_time = None # Reset start time for the next segment

# Close the database connection

conn.close()

Calculate the cumulative sailing time and the time spent at ports for each voyage segment.

import sqlite3

from datetime import datetime, timedelta

# Connect to the SQLite database

conn = sqlite3.connect('voyages.db')

cursor = conn.cursor()

# Fetch the data from the table

cursor.execute("SELECT id, event, dateStamp, timeStamp, voyage\_Id FROM voyages ORDER BY voyage\_Id, dateStamp, timeStamp")

rows = cursor.fetchall()

# Function to convert dateStamp and timeStamp to a precise UTC datetime

def convert\_to\_utc(dateStamp, timeStamp):

base\_date = datetime(1899, 12, 30) # Excel's epoch date

event\_date = base\_date + timedelta(days=dateStamp + timeStamp)

return event\_date

# Identify and segment different voyage stages

voyages = {}

for row in rows:

event\_id, event\_name, dateStamp, timeStamp, voyage\_Id = row

utc\_datetime = convert\_to\_utc(dateStamp, timeStamp)

if voyage\_Id not in voyages:

voyages[voyage\_Id] = []

voyages[voyage\_Id].append((event\_name, utc\_datetime))

# Calculate cumulative sailing time and time spent at ports

for voyage\_Id, events in voyages.items():

sailing\_time = timedelta()

port\_time = timedelta()

last\_eosp\_time = None

start\_time = None

for event\_name, event\_time in events:

if event\_name == 'SOSP':

if last\_eosp\_time:

port\_time += event\_time - last\_eosp\_time

start\_time = event\_time

elif event\_name == 'EOSP' and start\_time:

sailing\_time += event\_time - start\_time

last\_eosp\_time = event\_time

print(f"Voyage {voyage\_Id}:")

print(f" Cumulative Sailing Time: {sailing\_time}")

print(f" Time Spent at Ports: {port\_time}")

# Close the database connection

conn.close()

import sqlite3

from datetime import datetime, timedelta

import math

# Connect to the SQLite database

conn = sqlite3.connect('voyages.db')

cursor = conn.cursor()

# Fetch the data from the table

cursor.execute("SELECT id, event, dateStamp, timeStamp, voyage\_From, lat, lon, voyage\_Id FROM voyages ORDER BY voyage\_Id, dateStamp, timeStamp")

rows = cursor.fetchall()

# Function to convert dateStamp and timeStamp to a precise UTC datetime

def convert\_to\_utc(dateStamp, timeStamp):

base\_date = datetime(1899, 12, 30) # Excel's epoch date

event\_date = base\_date + timedelta(days=dateStamp + timeStamp)

return event\_date

# Haversine formula to calculate the distance between two lat/lon points

def haversine(lat1, lon1, lat2, lon2):

R = 6371.0 # Earth radius in kilometers

dlat = math.radians(lat2 - lat1)

dlon = math.radians(lon2 - lon1)

a = math.sin(dlat / 2)\*\*2 + math.cos(math.radians(lat1)) \* math.cos(math.radians(lat2)) \* math.sin(dlon / 2)\*\*2

c = 2 \* math.atan2(math.sqrt(a), math.sqrt(1 - a))

distance = R \* c

return distance

# Identify and segment different voyage stages

voyages = {}

for row in rows:

event\_id, event\_name, dateStamp, timeStamp, voyage\_From, lat, lon, voyage\_Id = row

utc\_datetime = convert\_to\_utc(dateStamp, timeStamp)

if voyage\_Id not in voyages:

voyages[voyage\_Id] = []

voyages[voyage\_Id].append((event\_name, utc\_datetime, lat, lon))

# Calculate cumulative sailing time, time spent at ports, and distances between ports

for voyage\_Id, events in voyages.items():

sailing\_time = timedelta()

port\_time = timedelta()

total\_distance = 0.0

last\_eosp\_time = None

start\_time = None

last\_position = None

for event\_name, event\_time, lat, lon in events:

if event\_name == 'SOSP':

if last\_eosp\_time:

port\_time += event\_time - last\_eosp\_time

start\_time = event\_time

if last\_position:

total\_distance += haversine(last\_position[0], last\_position[1], lat, lon)

elif event\_name == 'EOSP' and start\_time:

sailing\_time += event\_time - start\_time

last\_eosp\_time = event\_time

last\_position = (lat, lon)

print(f"Voyage {voyage\_Id}:")

print(f" Cumulative Sailing Time: {sailing\_time}")

print(f" Time Spent at Ports: {port\_time}")

print(f" Total Distance Traveled: {total\_distance:.2f} km")

# Close the database connection

conn.close()

1. <https://opposite-rudbeckia-e0f.notion.site/Financial-Record-33577389af014e819dfe6737d71ee496?pvs=4>

exports.handler = async (event) => {

try {

const body = JSON.parse(event.body);

// Validate the incoming JSON structure

if (!isValidTransaction(body)) {

return {

statusCode: 400,

body: JSON.stringify({ error: 'Invalid transaction data' })

};

}

// Process the transaction (you can implement your business logic here)

console.log('Received transaction:', body);

// Return success response

return {

statusCode: 200,

body: JSON.stringify({ message: 'Transaction processed successfully' })

};

} catch (error) {

console.error('Error processing transaction:', error);

return {

statusCode: 500,

body: JSON.stringify({ error: 'Internal server error' })

};

}

};

function isValidTransaction(data) {

// Implement your validation logic here

return (

data &&

data.transactionId &&

data.userId &&

data.transactionDetails &&

data.transactionDetails.amount &&

data.transactionDetails.currency &&

data.transactionDetails.transactionDate &&

data.transactionDetails.paymentMethod &&

data.transactionDetails.merchantDetails &&

data.transactionDetails.merchantDetails.merchantId &&

data.transactionDetails.merchantDetails.name &&

data.transactionDetails.merchantDetails.category &&

data.transactionDetails.merchantDetails.countryCode &&

data.userDetails &&

data.userDetails.firstName &&

data.userDetails.lastName &&

data.userDetails.email &&

data.userDetails.phone &&

data.userDetails.billingAddress &&

data.userDetails.billingAddress.street &&

data.userDetails.billingAddress.city &&

data.userDetails.billingAddress.state &&

data.userDetails.billingAddress.postalCode &&

data.userDetails.billingAddress.country &&

data.additionalInfo &&

data.additionalInfo.deviceIp &&

data.additionalInfo.userAgent

);

}