Übung 4: Merge-Konflikte und deren Auflösung

Lernziele

Durch das Absolvieren dieser Übung werden Sie folgende Bereiche praktizieren:

- Verstehen, was Merge-Konflikte verursacht
- Identifizieren und Analysieren von Merge-Konflikt-Markierungen
- Auflösen verschiedener Arten von Merge-Konflikten
- Verwendung von Merge-Strategien und -Tools
- Vermeiden von Konflikten durch gute Praktiken

Szenario

Sie leiten ein Data Science-Team, in dem mehrere Entwickler am gleichen Analytics-Projekt arbeiten. Konflikte sind unvermeidlich, wenn Teammitglieder die gleichen Dateien ändern. Diese Übung lehrt Sie, diese Konflikte professionell und effizient zu handhaben.

Setup

Initialisieren Sie das Analytics-Projekt:

```
# Projektverzeichnis erstellen
mkdir DataScienceProjekt
cd DataScienceProjekt

# Repository initialisieren
git init
git config user.name "Ihr Name"
git config user.email "ihre.email@beispiel.de"
git branch -M main
```

Teil 1: Projektgrundlage und erste Konflikte

Schritt 1: Basis-Projekt erstellen

Erstellen Sie die Grundlagendateien:

`projekt_config.py`:

```
# Data Science Projekt Konfiguration
import os

# Datenbankverbindungseinstellungen
DATABASE_HOST = "localhost"
DATABASE_PORT = 5432
DATABASE_NAME = "analytics_db"
```

```
DATABASE_USER = "analyst"

# Analyseparameter
ANALYSIS_START_DATE = "2024-01-01"
ANALYSIS_END_DATE = "2024-12-31"
CONFIDENCE_LEVEL = 0.95

# Ausgabeeinstellungen
OUTPUT_FORMAT = "csv"
CHART_STYLE = "seaborn"
```

datenanalyse.py:

```
# Haupt-Datenanalyse-Skript
import pandas as pd
import numpy as np
from datetime import datetime

def lade_kundendaten():
    """Lade Kundendaten aus Datenbank"""
    print("Lade Kundendaten...")
    # Platzhalter für Datenbankverbindung
    return pd.DataFrame()

def analysiere_kundentrends():
    """Analysiere Kundenverhaltenstrends"""
    daten = lade_kundendaten()
    print("Analysiere Kundentrends...")
    # Basis-Analyse Platzhalter
    return {"gesamtkunden": 1000}

def generiere_bericht():
    """Generiere Analysebericht"""
    ergebnisse = analysiere_kundentrends()
    print(f"Analyse abgeschlossen. {ergebnisse['gesamtkunden']} Kunden gefunden.")
    return ergebnisse

if __name__ == "__main__":
    generiere_bericht()
```

README.md:

```
# Data Science Analytics Projekt

## Überblick
Dieses Projekt analysiert Kundenverhalten und generiert Erkenntnisse für
Geschäftsentscheidungen.

## Teammitglieder
```

```
- Lead Analyst: [Ihr Name]

## Aktueller Status
- Projekteinrichtung: Abgeschlossen
- Datensammlung: In Bearbeitung
- Analyse: Geplant

## Features
- Kundenverhaltenanalyse
- Trend-Identifikation
- Automatisierte Berichterstattung
```

1. Erstellen Sie alle Dateien und committen Sie: "Initiale Projekteinrichtung mit Konfigurations- und Analyse-Framework"

Schritt 2: Entwickler A's Arbeit simulieren

1. Erstellen Sie einen Branch für Entwickler A: `git switch -c feature/datenbank-integration`

Ändern Sie `projekt_config.py`, um Datenbank-Anmeldedaten hinzuzufügen:

```
# Data Science Projekt Konfiguration
import os

# Datenbankverbindungseinstellungen
DATABASE_HOST = "prod-analytics-server.firma.com"
DATABASE_PORT = 5432
DATABASE_NAME = "analytics_db"
DATABASE_USER = "analyst"
DATABASE_PASSWORD = "sicheres_passwort_123"

# Analyseparameter
ANALYSIS_START_DATE = "2024-01-01"
ANALYSIS_END_DATE = "2024-12-31"
CONFIDENCE_LEVEL = 0.95

# Ausgabeeinstellungen
OUTPUT_FORMAT = "csv"
CHART_STYLE = "seaborn"
```

Ändern Sie `datenanalyse.py`, um echte Datenbankfunktionalität hinzuzufügen:

```
# Haupt-Datenanalyse-Skript
import pandas as pd
import numpy as np
from datetime import datetime
import psycopg2
import projekt_config as config
```

```
def lade_kundendaten():
    """Lade Kundendaten aus Datenbank"""
    print("Verbinde mit Produktionsdatenbank...")
    conn = psycopg2.connect(
        host=config.DATABASE_HOST,
        port=config.DATABASE_PORT,
        database=config.DATABASE_NAME,
        user=config.DATABASE_USER,
        password=config.DATABASE_PASSWORD
    )

    query = """
    SELECT kunden_id, registrierungsdatum, gesamtkaeufe, kundensegment
    FROM kunden
    WHERE registrierungsdatum >= %s AND registrierungsdatum <= %s
    """

    daten = pd.read_sql(query, conn, params=[config.ANALYSIS_START_DATE,
config.ANALYSIS_END_DATE])
    conn.close()
    return daten

def analysiere_kundentrends():
    """Analysiere Kundenverhaltenstrends"""
    daten = lade_kundendaten()
    print("Analysiere Kundentrends...")

    # Erweiterte Analyse
    segment_analyse = daten.groupby('kundensegment').agg({
        'gesamtkaeufe': ['count', 'mean', 'sum'],
        'kunden_id': 'count'
    }).round(2)

    return {
        "gesamtkunden": len(daten),
        "segmente": segment_analyse.to_dict()
    }

def generiere_bericht():
    """Generiere Analysebericht"""
    ergebnisse = analysiere_kundentrends()
    print(f"Analyse abgeschlossen. {ergebnisse['gesamtkunden']} Kunden gefunden.")
    print("Segment-Analyse abgeschlossen.")
    return ergebnisse

if __name__ == "__main__":
    generiere_bericht()
```

2. Committen Sie Entwickler A's Arbeit: "Produktions-Datenbankintegration und erweiterte Analytik hinzugefügt"

Schritt 3: Entwickler B's Arbeit simulieren

1. Wechseln Sie zurück zu main: `git switch main`
2. Erstellen Sie einen Branch für Entwickler B: `git switch -c feature/visualisierung-upgrade`

Ändern Sie `projekt_config.py`, um Visualisierungseinstellungen hinzuzufügen:

```
# Data Science Projekt Konfiguration
import os

# Datenbankverbindungseinstellungen
DATABASE_HOST = "localhost"
DATABASE_PORT = 5432
DATABASE_NAME = "analytics_db"
DATABASE_USER = "analyst"

# Analyseparameter
ANALYSIS_START_DATE = "2024-01-01"
ANALYSIS_END_DATE = "2024-12-31"
CONFIDENCE_LEVEL = 0.95

# Ausgabeeinstellungen
OUTPUT_FORMAT = "json"  # Von csv zu json geändert
CHART_STYLE = "plotly"   # Von seaborn zu plotly geändert
CHART_WIDTH = 1200
CHART_HEIGHT = 800
COLOR_PALETTE = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728"]

# Neue Visualisierungsparameter
ENABLE_INTERACTIVE_CHARTS = True
SAVE_CHARTS_TO_FILE = True
CHART_OUTPUT_DIR = "./charts"
```

Ändern Sie `datenanalyse.py`, um Visualisierungsfeatures hinzuzufügen:

```
# Haupt-Datenanalyse-Skript
import pandas as pd
import numpy as np
from datetime import datetime
import plotly.express as px
import plotly.graph_objects as go
import projekt_config as config

def lade_kundendaten():
    """Lade Kundendaten aus Datenbank"""
    print("Lade Kundendaten...")
    # Verwende Beispieldaten für Visualisierungsentwicklung
    np.random.seed(42)
    beispieldaten = pd.DataFrame({
        'kunden_id': range(1, 1001),
        'registrierungsdatum': pd.date_range('2024-01-01', periods=1000,
freq='D'),
```

```
        'gesamtkaeufe': np.random.lognormal(3, 1, 1000),
        'kundensegment': np.random.choice(['Premium', 'Standard', 'Basic'], 1000)
    })
    return beispieldaten

def erstelle_visualisierungen(daten):
    """Erstelle interaktive Visualisierungen"""
    print("Erstelle interaktive Visualisierungen...")

    # Kundensegment-Verteilung
    fig1 = px.pie(daten, names='kundensegment',
                  title='Kundensegment-Verteilung',
                  color_discrete_sequence=config.COLOR_PALETTE,
                  width=config.CHART_WIDTH,
                  height=config.CHART_HEIGHT)

    # Kaufbetrag-Verteilung nach Segment
    fig2 = px.box(daten, x='kundensegment', y='gesamtkaeufe',
                  title='Kaufverteilung nach Segment',
                  color='kundensegment',
                  color_discrete_sequence=config.COLOR_PALETTE,
                  width=config.CHART_WIDTH,
                  height=config.CHART_HEIGHT)

    if config.SAVE_CHARTS_TO_FILE:
        import os
        os.makedirs(config.CHART_OUTPUT_DIR, exist_ok=True)
        fig1.write_html(f"{config.CHART_OUTPUT_DIR}/segment_verteilung.html")
        fig2.write_html(f"{config.CHART_OUTPUT_DIR}/kauf_verteilung.html")

    if config.ENABLE_INTERACTIVE_CHARTS:
        fig1.show()
        fig2.show()

    return fig1, fig2

def analysiere_kundentrends():
    """Analysiere Kundenverhaltenstrends"""
    daten = lade_kundendaten()
    print("Analysiere Kundentrends...")

    # Erstelle Visualisierungen
    diagramme = erstelle_visualisierungen(daten)

    # Basis-Analyse
    analyse_ergebnisse = {
        "gesamtkunden": len(daten),
        "durchschnittliche_kaeufe": daten['gesamtkaeufe'].mean().round(2),
        "erstellte_diagramme": len(diagramme)
    }

    return analyse_ergebnisse

def generiere_bericht():
```

```
    """Generiere Analysebericht"""
    ergebnisse = analysiere_kundentrends()
    print(f"Analyse abgeschlossen. {ergebnisse['gesamtkunden']} Kunden gefunden.")
    print(f"Durchschnittliche Käufe: ${ergebnisse['durchschnittliche_kaeufe']}")
    print(f"{ergebnisse['erstellte_diagramme']} interaktive Diagramme erstellt.")
    return ergebnisse

if __name__ == "__main__":
    generiere_bericht()
```

3. Committen Sie Entwickler B's Arbeit: "Interaktive Visualisierungen und erweiterte Berichterstattung hinzugefügt"

Teil 2: Erstellen und Auflösen von Basis-Konflikten

Schritt 1: Erstes Feature zusammenführen

1. Wechseln Sie zu main: `git switch main`
2. Führen Sie Entwickler A's Arbeit zusammen: `git merge feature/datenbank-integration`
3. Das sollte sauber zusammengeführt werden, da main sich nicht geändert hat

Schritt 2: Ihren ersten Merge-Konflikt erstellen

1. Versuchen Sie, Entwickler B's Arbeit zusammenzuführen: `git merge feature/visualisierung-upgrade`
2. Sie sollten einen Merge-Konflikt bekommen! Git zeigt etwas wie:

```
Auto-merging datenanalyse.py
CONFLICT (content): Merge conflict in datenanalyse.py
Auto-merging projekt_config.py
CONFLICT (content): Merge conflict in projekt_config.py
Automatic merge failed; fix conflicts and then commit the result.
```

3. Überprüfen Sie den Status: `git status`

Schritt 3: Konflikt-Markierungen verstehen

Öffnen Sie `projekt_config.py` in Ihrem Texteditor. Sie sehen Konflikt-Markierungen:

```
# Data Science Projekt Konfiguration
import os

# Datenbankverbindungseinstellungen
DATABASE_HOST = "prod-analytics-server.firma.com"
DATABASE_PORT = 5432
DATABASE_NAME = "analytics_db"
DATABASE_USER = "analyst"
<<<<<<< HEAD
```

```
DATABASE_PASSWORD = "sicheres_passwort_123"
=======
>>>>>>> feature/visualisierung-upgrade

# Analyseparameter
ANALYSIS_START_DATE = "2024-01-01"
ANALYSIS_END_DATE = "2024-12-31"
CONFIDENCE_LEVEL = 0.95

# Ausgabeeinstellungen
<<<<<<< HEAD
OUTPUT_FORMAT = "csv"
CHART_STYLE = "seaborn"
=======
OUTPUT_FORMAT = "json"  # Von csv zu json geändert
CHART_STYLE = "plotly"   # Von seaborn zu plotly geändert
CHART_WIDTH = 1200
CHART_HEIGHT = 800
COLOR_PALETTE = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728"]

# Neue Visualisierungsparameter
ENABLE_INTERACTIVE_CHARTS = True
SAVE_CHARTS_TO_FILE = True
CHART_OUTPUT_DIR = "./charts"
>>>>>>> feature/visualisierung-upgrade
```

Markierungen verstehen:

- `<<<<<<< HEAD` - Start der aktuellen Branch (main) Änderungen
- `=======` - Trenner zwischen konfliktierenden Änderungen
- `>>>>>>> feature/visualisierung-upgrade` - Ende der eingehenden Branch-Änderungen

Schritt 4: Konfigurationskonflikte auflösen

Lösen Sie `projekt_config.py` auf, indem Sie das Beste aus beiden Versionen kombinieren:

```
# Data Science Projekt Konfiguration
import os

# Datenbankverbindungseinstellungen
DATABASE_HOST = "prod-analytics-server.firma.com"
DATABASE_PORT = 5432
DATABASE_NAME = "analytics_db"
DATABASE_USER = "analyst"
DATABASE_PASSWORD = "sicheres_passwort_123"

# Analyseparameter
ANALYSIS_START_DATE = "2024-01-01"
ANALYSIS_END_DATE = "2024-12-31"
CONFIDENCE_LEVEL = 0.95
```

```
# Ausgabeeinstellungen
OUTPUT_FORMAT = "json"  # JSON für bessere Datenstruktur verwenden
CHART_STYLE = "plotly"   # Plotly für interaktive Diagramme verwenden
CHART_WIDTH = 1200
CHART_HEIGHT = 800
COLOR_PALETTE = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728"]

# Visualisierungsparameter
ENABLE_INTERACTIVE_CHARTS = True
SAVE_CHARTS_TO_FILE = True
CHART_OUTPUT_DIR = "./charts"
```

Schritt 5: Code-Konflikte auflösen

Öffnen Sie nun `datenanalyse.py`. Dies wird komplexere Konflikte haben. Sie müssen kombinieren:

- Datenbankfunktionalität von Entwickler A
- Visualisierungsfunktionalität von Entwickler B
- Sicherstellen, dass Imports für beide Features funktionieren

Erstellen Sie eine aufgelöste Version, die beide Features kombiniert:

```
# Haupt-Datenanalyse-Skript
import pandas as pd
import numpy as np
from datetime import datetime
import os
import psycopg2
import plotly.express as px
import plotly.graph_objects as go
import projekt_config as config

def lade_kundendaten():
    """Lade Kundendaten aus Datenbank"""
    print("Verbinde mit Produktionsdatenbank...")
    try:
        conn = psycopg2.connect(
            host=config.DATABASE_HOST,
            port=config.DATABASE_PORT,
            database=config.DATABASE_NAME,
            user=config.DATABASE_USER,
            password=config.DATABASE_PASSWORD
        )

        query = """
        SELECT kunden_id, registrierungsdatum, gesamtkaeufe, kundensegment
        FROM kunden
        WHERE registrierungsdatum >= %s AND registrierungsdatum <= %s
        """

        daten = pd.read_sql(query, conn, params=[config.ANALYSIS_START_DATE,
```

```
config.ANALYSIS_END_DATE])
        conn.close()
        print(f"{len(daten)} Datensätze aus Datenbank geladen")
        return daten

    except Exception as e:
        print(f"Datenbankverbindung fehlgeschlagen: {e}")
        print("Verwende Beispieldaten für Entwicklung...")
        # Fallback zu Beispieldaten
        np.random.seed(42)
        beispieldaten = pd.DataFrame({
            'kunden_id': range(1, 1001),
            'registrierungsdatum': pd.date_range('2024-01-01', periods=1000,
freq='D'),
            'gesamtkaeufe': np.random.lognormal(3, 1, 1000),
            'kundensegment': np.random.choice(['Premium', 'Standard', 'Basic'],
1000)
        })
        return beispieldaten

def erstelle_visualisierungen(daten):
    """Erstelle interaktive Visualisierungen"""
    print("Erstelle interaktive Visualisierungen...")

    # Kundensegment-Verteilung
    fig1 = px.pie(daten, names='kundensegment',
                  title='Kundensegment-Verteilung',
                  color_discrete_sequence=config.COLOR_PALETTE,
                  width=config.CHART_WIDTH,
                  height=config.CHART_HEIGHT)

    # Kaufbetrag-Verteilung nach Segment
    fig2 = px.box(daten, x='kundensegment', y='gesamtkaeufe',
                  title='Kaufverteilung nach Segment',
                  color='kundensegment',
                  color_discrete_sequence=config.COLOR_PALETTE,
                  width=config.CHART_WIDTH,
                  height=config.CHART_HEIGHT)

    if config.SAVE_CHARTS_TO_FILE:
        os.makedirs(config.CHART_OUTPUT_DIR, exist_ok=True)
        fig1.write_html(f"{config.CHART_OUTPUT_DIR}/segment_verteilung.html")
        fig2.write_html(f"{config.CHART_OUTPUT_DIR}/kauf_verteilung.html")
        print("Diagramme in Dateien gespeichert")

    if config.ENABLE_INTERACTIVE_CHARTS:
        fig1.show()
        fig2.show()

    return fig1, fig2

def analysiere_kundentrends():
    """Analysiere Kundenverhaltenstrends"""
    daten = lade_kundendaten()
```

```
    print("Analysiere Kundentrends...")

    # Erstelle Visualisierungen
    diagramme = erstelle_visualisierungen(daten)

    # Erweiterte Analyse beide Ansätze kombinierend
    segment_analyse = daten.groupby('kundensegment').agg({
        'gesamtkaeufe': ['count', 'mean', 'sum'],
        'kunden_id': 'count'
    }).round(2)

    analyse_ergebnisse = {
        "gesamtkunden": len(daten),
        "segmente": segment_analyse.to_dict(),
        "durchschnittliche_kaeufe": daten['gesamtkaeufe'].mean().round(2),
        "erstellte_diagramme": len(diagramme)
    }

    return analyse_ergebnisse

def generiere_bericht():
    """Generiere Analysebericht"""
    ergebnisse = analysiere_kundentrends()
    print(f"Analyse abgeschlossen. {ergebnisse['gesamtkunden']} Kunden gefunden.")
    print(f"Durchschnittliche Käufe: ${ergebnisse['durchschnittliche_kaeufe']}")
    print(f"{ergebnisse['erstellte_diagramme']} interaktive Diagramme erstellt.")
    print("Segment-Analyse abgeschlossen.")
    return ergebnisse

if __name__ == "__main__":
    generiere_bericht()
```

Schritt 6: Merge abschließen

1. Stagen Sie die aufgelösten Dateien: `git add projekt_config.py datenanalyse.py`
2. Überprüfen Sie, dass Konflikte aufgelöst sind: `git status`
3. Schließen Sie den Merge ab: `git commit -m "Visualisierungsfeatures mit Datenbankintegration zusammengeführt"`
4. Überprüfen Sie den Merge: `git log --oneline --graph`

Teil 3: Weitere Konflikt-Szenarien und beste Praktiken

Schritt 1: Dokumentationskonflikte

1. Wechseln Sie zu einem neuen Branch: `git switch -c feature/dokumentation-update`

Ändern Sie `README.md` erheblich:

```
# Data Science Analytics Projekt

## Überblick
```

```
Dieses Projekt bietet umfassende Kundenverhaltenanalyse mit erweiterten
Visualisierungsmöglichkeiten und Produktions-Datenbankintegration.

## Teammitglieder
- Lead Analyst: [Ihr Name]
- Datenbankingenieur: Entwickler A
- Visualisierungsspezialist: Entwickler B

## Architektur
- **Datenbankschicht**: PostgreSQL mit sicheren Verbindungen
- **Analyse-Engine**: Pandas + NumPy für Datenverarbeitung
- **Visualisierung**: Plotly für interaktive Diagramme
- **Konfiguration**: Zentralisierte Konfigurationsverwaltung

## Aktueller Status
- Projekteinrichtung: Abgeschlossen ☑
- Datensammlung: Abgeschlossen ☑
- Datenbankintegration: Abgeschlossen ☑
- Visualisierungssystem: Abgeschlossen ☑
- Analyse-Pipeline: Abgeschlossen ☑

## Features
- Kundenverhaltenanalyse mit Segment-Aufschlüsselung
- Interaktive Datenvisualisierungen
- Automatisierte Berichtsgenerierung
- Produktions-Datenbankanbindung
- Fehlerbehandlung und Fallback-Mechanismen

## Verwendung
```bash
python datenanalyse.py
```

Ausgabe

- Analyseergebnisse im JSON-Format
- Interaktive HTML-Diagramme im ./charts Verzeichnis
- Konsolen-Berichterstattung mit Schlüsselmetriken

```
2. Committen Sie: "Größeres Dokumentations-Update mit Architektur und
Verwendungsdetails"

### Schritt 2: Gleichzeitige Dokumentationsänderungen erstellen
1. Wechseln Sie zurück zu main: `git checkout main`
2. Ändern Sie auch `README.md`, aber anders:
```markdown
# Data Science Analytics Projekt

## Überblick
Dieses Projekt analysiert Kundenverhalten und generiert Erkenntnisse für
Geschäftsentscheidungen.
```

```
## Teammitglieder
- Lead Analyst: [Ihr Name]

## Aktueller Status
- Projekteinrichtung: Abgeschlossen
- Datensammlung: Abgeschlossen
- Analyse: Abgeschlossen
- Visualisierungen: Abgeschlossen

## Features
- Kundenverhaltenanalyse
- Trend-Identifikation
- Automatisierte Berichterstattung
- Datenbankintegration
- Interaktive Diagramme

## Schnellstart
1. Datenbankeinstellungen in projekt_config.py konfigurieren
2. Ausführen: `python datenanalyse.py`
3. Diagramme im ./charts Verzeichnis anzeigen

## Abhängigkeiten
- pandas
- numpy
- psycopg2
- plotly

## Letzte Updates
- Produktions-Datenbank-Unterstützung hinzugefügt
- Interaktive Visualisierungen implementiert
- Erweiterte Fehlerbehandlung
```

3. Committen Sie: "README mit Schnellstart-Anleitung und Abhängigkeiten aktualisiert"

Schritt 3: Dokumentationskonflikte auflösen

1. Versuchen Sie zusammenzuführen: `git merge feature/dokumentation-update`
2. Sie bekommen Konflikte in README.md
3. Lösen Sie auf, indem Sie eine umfassende Version erstellen, die beinhaltet:
    - Die detaillierte Architektur vom Feature-Branch
    - Die Schnellstart-Anleitung von main
    - Alle Teammitglieder und Features
    - Beide Verwendungsanleitungen

Erstellen Sie eine aufgelöste README.md:

```
# Data Science Analytics Projekt

## Überblick
Dieses Projekt bietet umfassende Kundenverhaltenanalyse mit erweiterten
```

```
Visualisierungsmöglichkeiten und Produktions-Datenbankintegration.

## Teammitglieder
- Lead Analyst: [Ihr Name]
- Datenbankingenieur: Entwickler A
- Visualisierungsspezialist: Entwickler B

## Architektur
- **Datenbankschicht**: PostgreSQL mit sicheren Verbindungen
- **Analyse-Engine**: Pandas + NumPy für Datenverarbeitung
- **Visualisierung**: Plotly für interaktive Diagramme
- **Konfiguration**: Zentralisierte Konfigurationsverwaltung

## Aktueller Status
- Projekteinrichtung: Abgeschlossen ☑
- Datensammlung: Abgeschlossen ☑
- Datenbankintegration: Abgeschlossen ☑
- Visualisierungssystem: Abgeschlossen ☑
- Analyse-Pipeline: Abgeschlossen ☑

## Features
- Kundenverhaltenanalyse mit Segment-Aufschlüsselung
- Interaktive Datenvisualisierungen
- Automatisierte Berichtsgenerierung
- Produktions-Datenbankanbindung
- Fehlerbehandlung und Fallback-Mechanismen

## Schnellstart
1. Datenbankeinstellungen in projekt_config.py konfigurieren
2. Ausführen: `python datenanalyse.py`
3. Diagramme im ./charts Verzeichnis anzeigen

## Verwendung
```bash
python datenanalyse.py
```

Abhängigkeiten

- pandas
- numpy
- psycopg2
- plotly

Ausgabe

- Analyseergebnisse im JSON-Format
- Interaktive HTML-Diagramme im ./charts Verzeichnis
- Konsolen-Berichterstattung mit Schlüsselmetriken

Letzte Updates
```

- Produktions-Datenbank-Unterstützung hinzugefügt
- Interaktive Visualisierungen implementiert
- Erweiterte Fehlerbehandlung
- Umfassende Dokumentation

```
4. Schließen Sie den Merge ab: `git add README.md` und `git commit -m "Umfassendes
Dokumentations-Update zusammengeführt"`

## Teil 4: Konflikt-Vermeidung und -Verwaltung

### Schritt 1: Requirements-Datei-Konflikt erstellen
1. Erstellen Sie einen Branch: `git checkout -b feature/dependency-management`
2. Erstellen Sie `requirements.txt`:
```

pandas==1.5.3 numpy==1.24.3 psycopg2-binary==2.9.6 plotly==5.14.1 jupyter==1.0.0 matplotlib==3.7.1 seaborn==0.12.2

```
3. Committen Sie: "requirements.txt mit spezifischen Versionen hinzugefügt"
4. Wechseln Sie zu main: `git checkout main`
5. Erstellen Sie eine andere `requirements.txt`:
```

pandas>=1.5.0 numpy>=1.20.0 psycopg2-binary>=2.9.0 plotly>=5.0.0 scikit-learn==1.2.2 scipy==1.10.1

```
6. Committen Sie: "Requirements mit flexiblen Versionen und ML-Bibliotheken
hinzugefügt"

### Schritt 2: Requirements-Konflikte auflösen
1. Führen Sie zusammen: `git merge feature/dependency-management`
2. Lösen Sie den Konflikt auf, indem Sie eine umfassende Requirements-Datei
erstellen:
```

Datenverarbeitung

pandas>=1.5.0,<2.0.0 numpy>=1.20.0,<2.0.0

Datenbank

psycopg2-binary>=2.9.0,❤️.0.0

Visualisierung

plotly>=5.0.0,<6.0.0 matplotlib>=3.7.0,<4.0.0 seaborn>=0.12.0,<1.0.0

Entwicklung und Analyse

jupyter>=1.0.0,<2.0.0 scikit-learn>=1.2.0,<2.0.0 scipy>=1.10.0,<2.0.0

```
3. Schließen Sie den Merge mit guten Praktiken ab

## Überprüfung und Review

### Testen Sie Ihr Verständnis
Beantworten Sie diese Szenarien:

1. **Szenario**: Sie haben versehentlich `git reset --hard HEAD~10` ausgeführt und
eine Woche Arbeit verloren. Was tun Sie?

2. **Szenario**: Sie haben einen Feature-Branch gelöscht, der noch nicht
zusammengeführt war. Wie stellen Sie ihn wieder her?

3. **Szenario**: Sie haben sensible Daten committed und bereits gepusht. Was ist
Ihr Aktionsplan?

### Beherrschung der Befehle überprüfen
Überprüfen Sie, ob Sie diese Wiederherstellungsbefehle effektiv verwenden können:

```powershell
# Wiederherstellungsbefehle
git merge                    # Branches kombinieren
git status                   # Merge-Status und Konflikte überprüfen
git add                      # Aufgelöste Dateien stagen
git commit                   # Merge abschließen
git merge --abort            # Problematischen Merge abbrechen
git diff                     # Unterschiede zwischen Branches sehen
```

Erwartete Ergebnisse

Nach Abschluss dieser Übung sollten Sie:

- Merge-Konflikt-Indikatoren und -Markierungen erkennen
- Verschiedene Arten von Konflikten systematisch auflösen
- Verstehen, wann Änderungen von verschiedenen Branches behalten werden
- Wissen, wie Features von mehreren Entwicklern kombiniert werden
- Merge-Commits erfolgreich abschließen können
- Strategien zur Konfliktvorbeugung verstehen

Bewährte Praktiken gelernt
```

- Mit Teammitgliedern über überlappende Arbeit kommunizieren
- Neueste Änderungen vor dem Start neuer Features pullen
- Feature-Branches fokussiert und kurzlebig halten
- Zusammengeführten Code vor dem Pushen gründlich testen
- Aussagekräftige Commit-Nachrichten verwenden, besonders für Merge-Commits
- Merge-Tools für komplexe Konflikte in Betracht ziehen

Hinweis: In professionellen Umgebungen werden Konflikte oft durch Code-Review-Prozesse und Pair Programming gelöst, statt durch individuelle Auflösung.