Masterarbeit Tagebuch

Sven Burkhardt

Inhaltsverzeichnis

---

1 Validierung und Orts-Matching 13.04.2025

Erweiterung der Datenvalidierung und Fehlerausgabe im Transkriptions-Workflow

Kurzbeschreibung

Erweiterung des Validierungsmoduls für strukturierte Prüfung exportierter JSON-Dokumente. Verbesserung des Ortsabgleichs mit Groundtruth-Tabelle zur eindeutigen Zuordnung von Geonames- und Nodegoat-IDs.

Erledigte Aufgaben

Validierungsmodul (`validation_module.py`)

✓ Erweiterte Validierungslogik `validate_extended()` implementiert
✓ Prüfung auf Pflichtfelder: `recipient`, `creation_date`, `creation_place`
✓ Prüfung von `mentioned_places` auf Geonames- und Nodegoat-ID
✓ Ausgabe einer statistischen Fehlerübersicht (z.B. „Geonames-ID fehlt: 5×“)

Transkribus-Hauptskript

✓ Einbindung des erweiterten Validierungsmoduls
✓ Speichern von Fehlern pro Datei in strukturierter Liste
✓ Ausgabe der häufigsten Fehlerarten am Ende der Verarbeitung
✓ Fix für `UnboundLocalError` bei fehlgeschlagenem Dokumentexport

Ortsabgleich (Place-Matching)- Funktioniert noch nicht

✓ Lowercase + Strip für Eingabe und Vergleichswerte vereinheitlicht
✓ Threshold bei Fuzzy-Matching auf 75 gesetzt
✓ Fehlerbehandlung bei fehlenden oder ungültigen Wikidata-IDs (z.B. float statt string)
✓ Entfernung des Feldes `type` aus allen Place-Objekten und JSON-Exports
✓ Debug-Ausgabe zur Match-Qualität hinzugefügt

Nächste Schritte

❑ Validierungsfehler zusätzlich in CSV speichern
❑ Orte ohne IDs visuell markieren oder zur Nachprüfung kennzeichnen
❑ Schutzmechanismus gegen ID-Veränderung im LLM-Enrichment einbauen

Offene Fragen

❑ Wie bekomme ich die Groundtruth-Daten? ❑ Prüfung auf fehlerhafte Personenentitäten (z.B. „des“, „Vereinsführer“)

---

2 LLM-Anreicherung und Metadaten-Extraktion 12.04.2025

Integration von ChatGPT-API und Erweiterung der Metadaten-Extraktion

Kurzbeschreibung

Heute wurden die LLM-Anreicherungsskripte optimiert und die Metadatenextraktion aus JSON-Dateien weiterentwickelt. Die Skripte ermöglichen jetzt eine präzisere Erkennung von Entitäten und eine strukturierte Anreicherung der Transkribus-Daten mit zusätzlichen Metadaten durch den Einsatz von Large Language Models.

Erledigte Aufgaben

Optimierung von ChatGPT-API-picture-to-JSON_2024.py

✓ Code-Kommentare für bessere Dokumentation ergänzt
✓ API-Kostenkalkulation präzisiert
✓ Prompt-Engineering für die Analyse historischer Dokumente verbessert
Ergebnis: Zuverlässigere Extraktion von Metadaten aus Bildmaterial mit detaillierter Kostenkontrolle

Weiterentwicklung der CSV-Merger-Skripte

✓ Metadaten_CSV_Merger_V2.py mit erweiterter Normalisierungsfunktion implementiert
✓ Verbesserte Zusammenführung mehrerer CSV-Quellen
✓ Robustere Fehlerbehandlung bei der CSV-Verarbeitung hinzugefügt
Ergebnis: Umfassendere Gesamtübersicht über alle Akten mit konsistenter Nummerierung

Nächste Schritte

❑ Automatisierte Validierung der extrahierten Metadaten implementieren
❑ Named Entity Recognition für Militäreinheiten und Dienstgrade integrieren
❑ Dashboard zur Visualisierung der Metadaten-Qualität entwickeln

Offene Fragen

❑ Wie kann die Qualität der LLM-Extraktion objektiv bewertet werden?
❑ Skalierungsmöglichkeiten für größere Dokumentenmengen?

3 Integration mit Transkribus XML Export 11.04.2025

Erweiterung der Datenpipeline zur Anreicherung von Transkribus-XML-Dateien

Kurzbeschreibung

Das neue Modul `Transkribus_XML_Data_Enricher_with_CSV.py` wurde entwickelt, um XML-Dateien aus Transkribus mit strukturierten Metadaten aus CSV-Dateien anzureichern. Dieses Skript ermöglicht die Zusammenführung von manuell erstellten Metadaten und automatisch extrahierten Informationen, um vollständigere Dokumentbeschreibungen zu erstellen.

Erledigte Aufgaben

Entwicklung des XML-Enricher-Skripts

✓ XML-Parser für Transkribus-Dateien implementiert
✓ Matching-Algorithmus zwischen XML- und CSV-Daten entwickelt
✓ Automatische Anreicherung der XML-Dateien mit CSV-Metadaten
Ergebnis: Funktionierendes Skript zur Anreicherung von XML-Dateien mit zusätzlichen Metadaten

Integration des Finder-Tag-Systems

✓ `Image_Tags_to_list.py` zur Extraktion von macOS Finder-Tags erstellt
✓ Automatische Konvertierung von Finder-Tags in strukturierte CSV-Daten
✓ Bereinigung und Normalisierung der extrahierten Tag-Informationen
Ergebnis: Erfolgreiche Integration des visuellen Tagging-Systems in die Metadatenpipeline

Technische Details zur XML-Anreicherung

```
# XML-Namespace festlegen (wie in den Transkribus-Dateien)
NS = {"ns": "http://schema.primaresearch.org/PAGE/gts/pagecontent/2013-07-15"}

# CSV einlesen und für Matching vorbereiten
df_csv = pd.read_csv(csv_file_path, sep=";", dtype=str, on_bad_lines='skip')
df_csv = df_csv[df_csv["Akte_Scan"].str.contains("Akte", na=False)]

# Extraktion der Seitenzahl für eindeutiges Matching
df_csv["csv_page_number"] = df_csv["Akte_Scan"].apply(
    lambda x: re.search(r"_S(\d+)", x).group(1) if re.search(r"_S(\d+)", x) else None
)

# XML-Dateien mit CSV-Informationen anreichern
for xml_file in xml_files:
    tree = ET.parse(xml_path)
    root = tree.getroot()

    # Metadaten aus Transkribus extrahieren
    transkribus_meta = root.find(".//ns:TranskribusMetadata", NS)
    doc_id = transkribus_meta.get("docId", "").strip()
    page_id = transkribus_meta.get("pageId", "").strip()

    # Matching mit CSV-Daten
    csv_match = find_matching_csv_entry(doc_id, xml_page_number)

    # CSV-Daten in XML integrieren
    csv_elem = ET.Element("CSVData")
    if csv_match:
        for key, value in csv_match.items():
            child = ET.SubElement(csv_elem, key)
            child.text = value

    # Angereicherte XML speichern
    root.append(csv_elem)
    tree.write(output_path, encoding="utf-8", xml_declaration=True)
```

Nächste Schritte

❑ Integration der XML-Anreicherung in die bestehende Metadatenextraktion
❑ Entwicklung eines Validierungssystems für die angereicherten XML-Dateien
❑ Ausweitung der Extraktionsmöglichkeiten auf weitere Metadatenfelder

---

4 Integration von person_matcher.py mit transkribus_to_base_schema.py

09.04.2025

Verbesserung der Personenerkennung durch Nutzung einer gemeinsamen CSV-Referenz

Kurzbeschreibung

Die beiden Module person_matcher.py und transkribus_to_base_schema.py wurden überarbeitet, um eine konsistente Personenerkennung zu gewährleisten. Die zentrale Verbesserung ist die Nutzung der CSV-Datei mit Personenmetadaten als gemeinsame Ground Truth. Dies ermöglicht eine zuverlässigere Erkennung und Matching von Personennamen mit Variationen in der Schreibweise.

Erledigte Aufgaben

Integration von person_matcher.py

✓ person_matcher.py mit Funktionen für CSV-Einlesen und Fuzzy-Matching erweitert
✓ Schnittstelle für den Zugriff auf bekannte Personen standardisiert
✓ match_person-Funktion für die Suche in der CSV-Datei optimiert
Ergebnis: Vereinheitlichter Personenabgleich über beide Module hinweg

Überarbeitung von transkribus_to_base_schema.py

✓ Funktionen aus person_matcher.py integriert
✓ Gemeinsamen CSV-Pfad für Personenreferenzen eingerichtet
✓ Personenerkennungslogik verbessert, um Fuzzy-Matching zu nutzen
Ergebnis: Zuverlässigere Erkennung von Personennamen mit konsistenter Referenzierung

5 Objektorientierte Umstrukturierung der Datenextraktions-Pipeline

08.04.2025

Verbesserung des Skripts auf Basis der Empfehlungen von Claude Code

Kurzbeschreibung

Die bestehende Datenextraktionspipeline wurde grundlegend überarbeitet und auf eine objektorientierte Struktur umgestellt. Statt der bisherigen Dictionary-basierten Implementierung nutzen wir nun die in ‘document_schemas.py‘ definierten Klassen wie ‘BaseDocument‘, ‘Person‘, ‘Organization‘ und ‘Place‘. Diese Umstellung bringt erhebliche Vorteile bei der Datenvalidierung und -konsistenz mit sich.

Erledigte Aufgaben

Objektorientierte Umstrukturierung von transkribus_to_base_schema.py

✓ Umstellung der Datenstrukturen auf Klassen statt Dictionaries.
✓ Integration der Validierungsfunktionen mit Fehlerausgabe.
✓ Anpassung der JSON-Serialisierung mittels to_json()-Methode.
Ergebnis: Robusteres Skript mit automatischer Datenvalidierung und besserer Wartbarkeit.

Details zur Umstrukturierung

```
# Alter Ansatz:
result = create_base_schema(metadata_info, transcript_text)

# Neuer objektorientierter Ansatz:
from document_schemas import BaseDocument, Person, Place, Event, Organization

doc = BaseDocument(
    attributes=metadata_info,
```

```
    content_transcription=transcript_text,
    mentioned_persons=[Person(**p) for p in custom_data["persons"]],
    mentioned_organizations=[Organization(**o) for o in custom_data["organizations"]],
    mentioned_places=[Place(**pl) for pl in custom_data["places"]],
    mentioned_dates=custom_data["dates"]
)

# Mit Validierung
if not doc.is_valid():
    print(f"Fehler in Dokument: {doc.validate()}")
```

Vorteile der neuen Implementierung

- **Datenvalidierung:** Automatische Überprüfung auf strukturelle Korrektheit
- **Typsicherheit:** Verbesserte Erkennung von Fehlern bereits zur Programmierzeit
- **Objektorientierung:** Zusammengehörige Daten und Funktionen in einer Klasse
- **Bessere Wartbarkeit:** Klare Schnittstellen und Kapselung von Implementierungsdetails
- **Erweiterbarkeit:** Einfachere Anpassung an neue Anforderungen durch Vererbung

Technische Details zur Integration

```
# In person_matcher.py: CSV-Einlese-Funktion
def load_known_persons_from_csv(csv_path: str) -> List[Dict[str, str]]:
    """
    Lädt die bekannten Personen aus der CSV-Datei.
    """
    if not os.path.exists(csv_path):
        print(f"Warnung: CSV-Datei nicht gefunden: {csv_path}")
        return []

    try:
        df = pd.read_csv(csv_path, sep=";")
        persons = []

        # Extrahiere relevante Spalten
        for _, row in df.iterrows():
            forename = row.get("schema:givenName", "").strip()
            familyname = row.get("schema:familyName", "").strip()

            if forename or familyname:  # Mindestens ein Namensteil vorhanden
                person = {
                    "forename": forename,
                    "familyname": familyname,
                    "id": row.get("Lfd. No.", ""),
                    # Weitere Metadaten
                }
                persons.append(person)

        return persons
    except Exception as e:
        print(f"Fehler beim Laden der CSV-Datei: {e}")
        return []

# Verbesserte person_matcher Funktion mit CSV-Standardwert
def match_person(
    person: Dict[str, str],
    candidates: List[Dict[str, str]] = None,
    threshold: int = 70
) -> Tuple[Optional[Dict[str, str]], int]:
    """
    Match a person against known persons from CSV (if no candidates provided)
    """
    if not person:
        return None, 0
```

```
    # Wenn keine Kandidaten angegeben, nutze CSV-Daten
    if candidates is None:
        candidates = KNOWN_PERSONS
```

Nächste Schritte

❑ Testen der verbesserten Personenerkennung mit einem größeren Datensatz
❑ Entwicklung von Methoden zur automatischen Zusammenführung von Personenduplikaten
❑ Integration fortgeschrittener NER-Methoden für noch bessere Personenerkennung
❑ Überprüfung aller generierten JSON-Dateien auf Validität

Offene Fragen

❑ Wie lässt sich die Konsistenz zwischen der CSV-Datei und extrahierten Personen automatisch überprüfen?
❑ Sollen Schwellwerte für das Fuzzy-Matching je nach Dokumenttyp angepasst werden?

6 Implementierung der Datenextraktions-Pipeline 03.04.2025

Diese Implementierung wurde mit Unterstützung von Claude Code und Sorin von RISE durchgeführt. Erster Einsatz von Claude Code

Kurzbeschreibung

Implementierung der Datenkonvertierungspipeline vom Transkribus XML-Format in das projektspezifische JSON-Basisschema. Es wurden zwei Python-Module entwickelt: ein Schema-Modul und ein Konvertierungsmodul, die zusammen die Grundlage für die strukturierte Datenhaltung bilden.

Erledigte Aufgaben

Definition des Basis-Schemas

✓ Erstellung des ***document_schemas.py*** Moduls mit Klassen für verschiedene Dokumenttypen.
Ergebnis: Implementierung der Basisklassen `Person`, `Organization`, `Place`, `Event` und `BaseDocument` sowie spezialisierte Klassen für `Brief`, `Postkarte` und `Protokoll` mit Validierungsfunktionen.

Konvertierungs-Pipeline

✓ Implementierung des ***transkribus_to_base_schema.py*** Skripts für die XML-Verarbeitung.
Ergebnis: Vollständige Pipeline für die Extraktion von Metadaten und Text aus Transkribus XML-Dateien sowie Konvertierung in das JSON-Basisschema mit automatischer Erkennung von benannten Entitäten (Personen, Organisationen, Orte, Daten).

Nächste Schritte

❑ Integration der Tag-Regeln aus der Transkribus-Anleitung in die Konvertierungspipeline.
❑ Erweitern der automatischen Metadatenextraktion für komplexere Inhaltsanalysen.
❑ Ausführen der Pipeline auf dem vollständigen Datensatz und Prüfung der Ergebnisse.

Offene Fragen

❑ Sollen zusätzliche spezialisierte Dokumenttypen implementiert werden?
❑ Wie können Entitätsreferenzen zwischen verschiedenen Dokumenten hergestellt werden?
Diese Implementierung wurde mit Unterstützung von Claude Code und Sorin von RISE durchgeführt.

---

7 Festlegung Tagging Regeln 11.03.2025

Kurzbeschreibung

Hier werden die Tagging Regeln in Transkribus beschrieben

Personen `person`

Mit dem Tag `person` sollen alle strings getaggt werden, die eine direkte Zuordnung eier Person ermöglichen.

☞ Beispiel 1: Vereinsführer, Alfons, Zimmermann, Alfons Zimmermann, Z. A.Zimmermann, Herr Zimmermann, Herr Alfons Zimmermann, etc
☞ Beispiel 2: Gauleitung, Gauleiter, Gauleiter Max Mustermann, Gauleiter M., Gauleiter Mustermann, etc
☞ Beispiel 3: Oberlehrer, Chorleiter, etc, wenn Ort, Name oder Organisation bekannt.

Signaturen `signature`

Mit dem Tag `signature` sollen alle Strings getaggt werden, die eine handschriftliche Unterschrift darstellen. Dieses Tag dient dazu, Unterschriften von klar erkennbaren Personennamen im Fließtext zu unterscheiden.

☞ Beispiel 1: **Eindeutig lesbare Signaturen** werden direkt getaggt: `<signature>R. Weiss</signature>`.

☞ Beispiel 2: **Teilweise unleserliche Signaturen** werden mit dem Tag `unclear` innerhalb von `signature` markiert: `<signature>R. We<unclear>[...]</unclear></signature>`.

☞ Beispiel 3: **Wenn nur ein Teil des Namens lesbar ist, aber eine Identifikation unsicher bleibt**, sollte die Unterschrift vollständig im Tag `unclear` innerhalb von `signature` stehen: `<signature><unclear>Unleserlich</unclear></sig`

☞ Beispiel 4: **Wenn eine Signatur einer bekannten Person zugeordnet werden kann, aber nicht vollständig lesbar ist, bleibt die Signatur erhalten und wird nicht als „Person"getaggt**: `<signature>A. Zimm<unclear>[...]</unclear></si`

Organisationen `organization`

Mit dem Tag `organization` sollen alle Strings getaggt werden, die eine direkte Zuordnung einer Organisation ermöglichen.

☞ Beispiel 1: Männerchor Murg, Verein Deutscher Arbeiter (V.D.A.), Murgtalschule, etc.
☞ Beispiel 2: Abkürzungen, wenn sie eine Organisation eindeutig bezeichnen, z.B. V.D.A., NSDAP, STAGMA, etc.
☞ Beispiel 3: Wenn der Organisationsname mit einer Positionsangabe kombiniert ist, wird nur der Organisationsname getaggt: „<person>Vereinsleiter <person>des **<organisation>Männerchor Murg</organisation>**".

Orte `place`

Mit dem Tag `place` sollen alle Strings getaggt werden, die sich auf einen geografischen Ort beziehen.

Hintergund ist Fehlervermeidung: Manche Organisationen haben mehrere Standorte (z. B. "Badischer Sängergau"kann in Karlsruhe oder Mannheim sein). Ein explizites Tagging vermeidet Verwechslungen. Flexibilität: Falls sich eine Organisation an mehreren Orten befindet oder in einem bestimmten Kontext mit einem anderen Ort assoziiert wird, bleibt die Information strukturiert erhalten.

☞ Beispiel 1: Murg (Baden), Freiburg, Berlin, Murgtal, Schwarzwald, etc.
☞ Beispiel 2: Orte mit näherer Bestimmung, z.B. „bei Berlin", „im Murgtal", "Mayerhof Nebenzimmer"werden getaggt, und die nähere Bestimmung muss innerhalb des Tags. Beispiel: „ **<place>im Murgtal</place>**".
☞ Beispiel 3: Adressen oder Ortsangaben in Verbindung mit Organisationen, z.B. „<organisation>Universität **<place>Basel</place>** </organisation>öder „Postfach 6, **<place>Murg </place>**".

Datum `date`

Mit dem Tag `date` werden alle expliziten Datumsangaben markiert und bereits im Tag um das format dd.mm.yyyy ergänzt.

☞ Beispiel 1: 9. Oktober 1940, 20.10.1940, den 3. Mai 1938, etc.
☞ Beispiel 2: Relative Datumsangaben („gestern", „letzten Freitag") werden getaggt.
☞ Beispiel 3: Falls ein Datum in Kombination mit einem Ort steht, wird nur das Datum getaggt: „**<place>Murg**

(Baden)</place>, den **<date>9. Oktober 1940</date>**".

Abkürzungen `abbrev`

Mit dem Tag `abbrev` werden alle Abkürzungen getaggt, die für eine eindeutige Entität stehen.

☞ Beispiel 1: Dr., Prof., St., Hr., Frl., Dipl.-Ing., etc.
☞ Beispiel 2: Organisationskürzel, wenn sie eindeutig sind: „**<abbrev>V.D.A.</abbrev>**".
☞ Beispiel 3: Falls eine ausgeschriebene Variante im selben Dokument vorhanden ist, bleibt die Abkürzung getaggt: **<person><abbrev>Dr.</abbrev>Weiß</person>**

Unclear `unclear`

Mit dem Tag `unclear` werden unleserliche oder schwer entzifferbare Textstellen markiert.

☞ Beispiel 1: Unklare Zeichen oder fehlende Buchstaben: „Er wohnte in **<unclear>[...]<unclear>**".
☞ Beispiel 2: Teilweise lesbare Wörter: „**<place>Frei<unclear>[...]<unclear><place>**".

Sic `sic`

Mit dem Tag `sic` werden Wörter markiert, die absichtlich in einer falschen oder ungewöhnlichen Schreibweise beibehalten werden.

☞ Beispiel 1: Offensichtliche Tippfehler, wenn sie im Originaltext so vorkommen: „Er hatt **<sic>einen</sic>** große Freude."
☞ Beispiel 2: Veraltete oder falsche Schreibweisen: „**<sic>Feber</sic>**"für Februar.
☞ Beispiel 3: Falls eine Korrektur notwendig ist, kann sie als Kommentar ergänzt werden.

8 Überblick Hilfscript CHatGPT_Api_TranskribusXML_to_JSONv3.py

06.03.2025

Dieses Skript verarbeitet XML-Dateien aus einer definierten Ordnerstruktur und sendet deren Inhalt an die OpenAI-API zur automatischen Analyse und Extraktion relevanter Metadaten. Das generierte Ergebnis wird als JSON-Datei gespeichert.

Das Skript ist speziell auf historische Dokumente (z. B. aus dem Männerchor Murg Corpus, 1925-1945) zugeschnitten. Es analysiert und strukturiert die Daten, um relevante Informationen wie Autoren, Empfänger, Orte, Ereignisse und Zeitangaben zu extrahieren.

9 Voraussetzungen & Einrichtung

9.1 API Key setzen

Bevor das Skript ausgeführt wird, muss der OpenAI-API-Schlüssel als Umgebungsvariable gesetzt werden. Dies kann dauerhaft in der /.zshrc-Datei erfolgen:

```
export OPENAI\_API\_KEY=''sk-...''
source ~/.zshrc
```

Alternativ kann das Skript mit VS Code gestartet werden, indem es aus dem Terminal mit folgendem Befehl aufgerufen wird:

```
code .
```

9.2 Python-Bibliotheken installieren

Folgende Bibliotheken werden benötigt:

```
pip install openai
```

folgende Module werden gebraucht:

```
import json
import os
import xml.etree.ElementTree as ET
import openai
import re
import time
```

10 Funktionsweise

Das Skript ist in folgende Hauptbestandteile gegliedert:

10.1 Basis-Einstellungen

- Zähler zur Erfassung der verarbeiteten Dateien und Token-Kosten.
- API-Schlüssel wird aus der Umgebungsvariable geladen.
- Pfad-Definitionen für Input- und Output-Verzeichnisse.

10.2 Verzeichnisstruktur durchlaufen

Das Skript iteriert über alle 7-stelligen Ordner im Basisverzeichnis (base_input_directory). In diesen Ordnern sucht es nach Unterordnern mit Namen Akte_xxx_pdf oder Akte_xxx, die wiederum einen page-Ordner enthalten.

Falls kein page-Ordner gefunden wird, wird die Verarbeitung dieser Akte übersprungen.

10.3 Verarbeitung der XML-Dateien

Jede XML-Datei im page-Ordner wird eingelesen und analysiert:

- Seitennummer extrahieren: Die Dateinamen haben das Muster p001.xml, p002.xml usw.
- XML-Daten einlesen: Nutzung von xml.etree.ElementTree zur Extraktion von TranskribusMetadata und TextEquiv-Daten.
- Fehlermanagement: Falls eine Datei nicht geparst werden kann oder keinen verwertbaren Text enthält, wird sie übersprungen.

10.4 Strukturierung der Daten in JSON

Die extrahierten Informationen werden in einem JSON-Format gespeichert. Dazu gehören:

- Metadaten (Dokument-ID, Seiten-ID, Bild- und XML-URL)
- Autor und Empfänger mit Name, Rolle und zugehöriger Organisation
- Erwähnte Personen, Organisationen, Ereignisse und Orte
- Dokumentart (z. B. Brief, Protokoll, Rechnung)
- Dokumentformat (z. B. Handschrift, maschinell, mit Unterschrift, Bild)

10.5 API-Anfrage an OpenAI

Ein Prompt wird erstellt, um die Inhalte durch die OpenAI-API analysieren zu lassen. Dabei wird explizit vorgegeben:

- Die historische Relevanz der Dokumente (1925-1945).
- Die Aufgabenstellung (Identifikation von Dokumenttypen, Metadaten und Inhalten).
- Die genaue Formatierung der JSON-Antwort.

11 Fehlerbehandlung

Das Skript enthält mehrere Mechanismen zur Fehlerbehandlung:

- Fehlermeldungen beim XML-Einlesen (try-except beim Parsen)
- Fehlermeldungen bei API-Anfragen (try-except um den OpenAI-Aufruf)
- Fehlermeldungen bei JSON-Speicherung (try-except beim Schreiben der Datei)
- Logging von Problemen (print(f"Fehler beim Parsen der API-Antwort: e "))

Falls Fehler auftreten, werden sie ausgegeben und das Skript setzt die Verarbeitung der nächsten Datei fort, anstatt komplett abzubrechen.

12 Fazit

Dieses Skript automatisiert die Verarbeitung von XML-Dokumenten, extrahiert deren Inhalte und strukturiert die Daten in JSON-Format, das von OpenAI-API analysiert wird. Die Ergebnisse werden gespeichert und abschließend statistisch ausgewertet. Durch Fehlerbehandlung und Logging wird sichergestellt, dass auch bei Problemen das Skript robust bleibt. Dokumentation geschrieben mit ChatGPT.

---

13 Tagging der JPGEGs im AppleFinder 26.10.24

Kurzbeschreibung

Überlegung: JPEGs sollen bereits im Apple Finder mit Tags versehen werden, um eine effiziente, automatisierte Transkription der Chorunterlagen des Männerchors Murg zu ermöglichen. Geplant ist die Kombination von ChatGPT und Transkribus zur Erkennung unterschiedlicher Dokumententypen. Ein Tag-System, bestehend aus **„Maschinell"** für maschinengeschriebene und **„Handschrift"** für handschriftliche Dokumente, gewährleistet die gezielte Zuordnung zur jeweils geeigneten OCR-Software *(Maschinenschrift mit ChatGPT, Handschrift mit Transkribus "German Giant").*

Dokumente, die sowohl maschinell erstellten Text als auch handschriftliche Elemente enthalten, werden entsprechend ihrer Hauptinformationsgehalt getaggt. Zusätzlich erhalten alle Dokumente mit Unterschriften den Tag **„Unterschrift"**, um eine gezielte Verarbeitung dieser Elemente sicherzustellen.

13.1 AppleFinder Tags

- **Handschrift**
- **Maschinell**
- **mitUnterschrift**
- **Bild**

Erledigte Aufgaben

Handschriften tagging

✓ taggen ● **Handschrift** in AppleFinder.
✓ taggen ● **Maschinell** in AppleFinder.
✓ taggen ● **Bild**
✓ taggen ● **mitUnterschrift**

Ergebnis: Handschriften, Maschinell, Bilder und alle handschriftlichen Unterschriften getaggt

Nächste Schritte

❑ Skripte schreiben, um maschinelle Text zu extrahieren ❑ Transkribus für Handschriftliches anschmeißen.
❑ Nach Gemeinsamkeiten in den Texten suchen, um automatisierte Abfrage für ChatGPT zu erstellen.
❑ Ggf. Aufteilung in unterschiedliche Korpora (Briefe handschr. Briefe schreibmaschine, Zeitungsunterlagen.)
❑ Transkribus für Handschriften verwenden.

Offene Fragen

❑ Sollen die Bilder gelöscht werden?

---

14 JPG Datenbereinigung - leere Seiten löschen 25.10.2024

JPG Datenbereinigung

Alle JPGs ohne Inhalt, also beispielsweise Rückseiten, werden gelöscht. Regel: sobald etwas handschriftlich oder gedruckt auf einer Seite steht, bleibt es erhalten. Im Moment sind auch Bilder (Bsp. Postkarten inbegriffen.Bilder mit Taggs

14.1 Anmerkung

Geschichte/Chronik/Gründung des Männerchors in Akte 323

Erledigte Aufgaben

JPG Datenbereinigung

✓Alle JPGs ohne Inhalt, also beispielsweise Rückseiten, werden gelöscht.
Ergebnis: Reiner JPG Korpus mit Schriftgut, aber auch Bildern (bspw. Postkarten)

Nächste Schritte

❑ Nach Gemeinsamkeiten in den Texten suchen, um automatisierte Abfrage für ChatGPT zu erstellen.
❑Ggf. Aufteilung in unterschiedliche Korpora (Briefe handschr. Briefe schreibmaschine, Zeitungsunterlagen.)
❑ Transkribus für Handschriften verwenden.

Offene Fragen

❑ Sollen die Bilder gelöscht werden?
❑ Handschriftliche, maschinengeschriebene und gemischte Daten taggen? Ggf. erst später mit ChatGPT.
❑ Transkribus für Handschriften verwenden?

---

15 Datennormalisierung PDF zu JPEG 24.10.2024

Kurzbeschrieb

Heute zwei Python-Skripte zur Normalisierung der Akten geschrieben:

Erledigte Augaben

PDF zu JPG Konvertierung

✓ Skript ***JPEG-to-PDF.py*** geschrieben.
Ergebnis Alle PDF-Seiten in JPGs umgewandelt, Dateinamen mit Seitenzahlen formatiert.

Prüfung der Aktennummern

✓ Skript ***Check-if-all-files-complete.py*** geschrieben.
Ergebnis: Überprüft, ob Akten von 001 bis 425 vorhanden sind. Alle Akten sind vollständig in JPG umgewandelt.

Nächste Schritte

❑ Daten für OCR-Bereinigung vorbereiten, leere Seiten manuell entfernen.

Offene Fragen

❑ Handschriftliche, maschinengeschriebene und gemischte Daten taggen? Ggf. erst später mit ChatGPT.
