

Navigation

Ansicht auswählen:

- 1 Datensubset laden
- 2 Daten bereinigen
- 3 Tokenisierung
- 4 Statistische Analyse
- 5 Word Embedding
- 6 Model Evaluation
- 7 Text Classification
- 8 Text-Generierung

1 Kapitel 1 – Dataset Loader: Genius Song Lyrics (Hugging Face)

Dataset Loader: Genius Song Lyrics (Hugging Face)

Data Source: <https://huggingface.co/datasets/sebastiandizon/genius-song-lyrics>

Die ursprüngliche Genius Song Lyrics Dataset enthält ca. 2.76 Millionen Songs (~ 9 GB CSV).

Um leichtgewichtig experimentieren zu können, erlaubt dieses Skript das Herunterladen und Speichern eines kleineren zufälligen Subsets (z.B. 1%) als lokale CSV-Datei.

Hinweis: Dieser Abschnitt dokumentiert die Schritte aus dem zugehörigen Notebook `load-data-subset.ipynb`. Alle Berechnungen wurden dort ausgeführt. Die Streamlit-App lädt lediglich die erzeugten Dateien und visualisiert die Ergebnisse.

1. Preparations

1.1 Load original Dataset from Hugging Face

Es wird nicht die komplette CSV geladen, sondern das Dataset über Hugging Face Datasets.

Standardmäßig lädt `load_dataset` hier nur die Metadaten + Zugriff auf den `train`-Split.

```
dataset = load_dataset("sebastiandizon/genius-song-lyrics", split="train")
```

1.3 Configuration

- Definiere den Prozentsatz des Subsets (z.B. 1%, 5%, 10%).
- Beachte: Hugging Face akzeptiert bei `split`-Notation nur Ganzahlen.
- Legt Ausgabeverzeichnis und Dateinamen fest.
- Erstelle das Verzeichnis, falls es noch nicht existiert.

```
subset_fraction = 1

subset_size = int(len(dataset) * subset_fraction / 100)

output_dir = "data/raw"
os.makedirs(output_dir, exist_ok=True)
```

```
output_path = os.path.join(output_dir, f"lyrics_subset_{subset_fraction}pct.csv")
```

2. Load and Save Subset

2.1 Load Subset of Dataset

- Setze einen **Seed** für Reproduzierbarkeit.
- Mische den Datensatz zufällig.
- Wähle die ersten `subset_size` Einträge als Subset.

```
dataset = dataset.shuffle(seed=42)  
  
dataset_small = dataset.select(range(subset_size))
```

2.2 Convert to pandas DataFrame

Konvertiere das Subset in ein `pandas.DataFrame`.

```
df = dataset_small.to_pandas()
```

2.3 Save Subset locally

Speichere das Subset als CSV-Datei im definierten Ausgabeverzeichnis.

```
df.to_csv(output_path, index=False)
```



Datensubset laden Resultat



Dataset-Infos

- Songs: 51,348
- Artists: 39,461
- Genres: 6
- Quelle: `data/raw/lyrics_subset_1pct.csv`



Vorschau

	title	tag	artist	year	views	features	lyrics		id	language_cld3	langu
0	2 Is Better 棍子	rap	Chris Travis	2017	4437	{}	[Intro] Bitch I'm clean Two sticks like Chow Mein Two sticks like Chow Mein Bitch I'm	3036329	en		en
1	Scottie	rap	KrJ	2012	89	{}	My old girl left me on her old bull shit So I play it off like Pippen on my old Bull shit...f	72180	en		en
2	Pirate Password	rock	The never land pirate band	2011	175	{}	[Intro: spoken] Avast there matey haha If a pirate asks ya for the password Yohoho is \	2122100	en		en
3	Indri	rock	Puta Volcano	2015	14	{}	Just throw a glimpse under the shell Ghostly voices shrill ecstatic screams Now he's g	6889288	en		en
4	Maps	misc	ANBARDA	2018	4	{}	[Verse 1] I miss the taste of a sweeter life I miss the conversation I'm searching for a s	3735887	en		en

🎵 Genre-Verteilung

GENRE DISTRIBUTION

pop: 21,438 songs (41.75%)

rap: 17,175 songs (33.45%)

rock: 8,001 songs (15.58%)

rb: 1,894 songs (3.69%)

misc: 1,860 songs (3.62%)

country: 980 songs (1.91%)

Navigation

Ansicht auswählen:

- 1 Datensubset laden
- 2 Daten bereinigen
- 3 Tokenisierung
- 4 Statistische Analyse
- 5 Word Embedding
- 6 Model Evaluation
- 7 Text Classification
- 8 Text-Generierung

2 Kapitel 2 – Data Cleaning: Genius Song Lyrics Subset

Purpose:

Bereinigung und Vorverarbeitung der Songtexte für Analysen.

Entfernt werden u.a.:

- Metadaten-Tags (z.B. `[Intro]`, `[Verse]`)
- Zeilenumbrüche (`\n`)
- überflüssige Leerzeichen

Ziel ist eine saubere Textspalte, die sich für **NLP** und **statistische Analysen** eignet.

Hinweis: Dieser Abschnitt dokumentiert die Schritte aus dem zugehörigen Notebook `data-cleaning.ipynb`. Die Bereinigung der Lyrics wurde vollständig im Notebook durchgeführt. Die Streamlit-App lädt lediglich die dort erzeugte bereinigte CSV-Datei und stellt ausgewählte Ergebnisse dar – ohne die Daten erneut zu bereinigen.

1. Dataset Overview

1.1 Load Dataset

Es wird das im **Kapitel 1** erzeugte Datensubset geladen, welches die rohen Lyrics enthält.

```
input_dir = "data/raw"  
input_path = os.path.join(input_dir, "lyrics_subset_1pct.csv")  
  
df = pd.read_csv(input_path)
```

2. Data Cleaning

2.1 Problem

Eine Vorschau der rohen Lyrics zeigt typische Probleme:

- Metadaten-Tags wie `[Intro]`, `[Verse]`, `[Hook]`
- Zeilenumbrüche `\n`
- Mehrfache bzw. führende/abschließende Leerzeichen

Diese Elemente erschweren spätere NLP-Analysen und müssen daher bereinigt werden.

```
0    [Intro]\nBitch I'm clean\nTwo sticks like Chow...
1    My old girl left me on her old bull shit\nSo I...
2    [Intro: spoken]\nAvast there matey haha\nIf a ...
3    Just throw a glimpse under the shell\nGhostly ...
4    [Verse 1]\nI miss the taste of a sweeter life\...
Name: lyrics, dtype: object
```

2.2 Define and Apply Cleaning Function

Die Cleaning-Funktion bereitet den Text in drei Schritten vor:

1. Entfernen aller Inhalte zwischen eckigen Klammern

```
re.sub(r'\[.*?\]', '', text)
```

Dieser Ausdruck löscht jeden Abschnitt, der zwischen `[` und `]` steht – inklusive des enthaltenen Textes. Dadurch verschwinden z. B. Annotationen, Quellen, Zeitstempel oder andere Meta-Informationen.

2. Ersetzen von Zeilenumbrüchen durch Leerzeichen

```
text.replace('\n', ' ')
```

Zeilenumbrüche werden in Leerzeichen umgewandelt, damit der Text eine durchgehende Linie bildet und besser weiterverarbeitet werden kann.

3. Reduzieren mehrfacher Leerzeichen & finales Formatieren

```
re.sub(r'\s+', ' ', text).strip()
```

Mehrere aufeinanderfolgende Leerzeichen werden zu einem einzigen zusammengefasst.

Gleichzeitig entfernt `.strip()` führende und nachfolgende Leerzeichen.

Das Ergebnis ist ein sauberer, kompakter Text ohne unnötige Abstände.

Endresultat: Der Text ist frei von Klammerinhalten, hat keine Zeilenumbrüche mehr und enthält nur noch einheitliche Leerzeichen – optimal zur weiteren Analyse oder NLP-Verarbeitung.

```
def clean_lyrics(text):
    text = re.sub(r'\[.*?\]', '', text)
    text = text.replace(
        '\n', ' ')
    text = re.sub(r'\s+', ' ', text).strip()
    return text

df["lyrics_clean"] = df["lyrics"].apply(clean_lyrics)
```

2.3 Preview Cleaned Lyrics

Nach Anwendung der Cleaning-Funktion sind:

- Metadaten-Tags (z.B. `[Intro]`, `[Verse]`) entfernt
- Zeilenumbrüche `\n` durch Leerzeichen ersetzt
- doppelte oder mehrfache Leerzeichen bereinigt

```
df = df.drop(columns=['lyrics'])
df = df.rename(columns={'lyrics_clean': 'lyrics'})
```

3. Save cleaned Data

3.1 Configuration

- Definiere Ausgabeverzeichnis und Dateinamen.
- Erstelle das Verzeichnis, falls es noch nicht existiert.

```
output_dir = "data/processed"
os.makedirs(output_dir, exist_ok=True)

output_path = os.path.join(output_dir, "lyrics_subset_1pct_clean.csv")
```

3.2 Prepare Data for Saving

Vor dem Speichern wird:

- die ursprüngliche Spalte `lyrics` entfernt
- `lyrics_clean` in `lyrics` umbenannt, sodass die bereinigten Texte im Feld `lyrics` liegen.

```
df = df.drop(columns=["lyrics"])

df = df.rename(columns={"lyrics_clean": "lyrics"})
```

📁 Bereinigte Daten

📌 Basis-Infos (bereinigtes Subset)

- Anzahl Zeilen (Songs): 51,348
- Spalten: title, tag, artist, year, views, features, id, language_cld3, language_ft, language, lyrics
- Quelle: `data/clean/lyrics_subset_1pct_clean.csv`

00 Vorschau der bereinigten Lyrics

	title	tag	artist	year	views	features	id	language_cld3	language_ft	language	lyrics
0	2 Is Better 棍子	rap	Chris Travis	2017	4437	{}	3036329	en	en	en	Bitch I'm clean Two sticks like Chow Mein Two sticks like Chow Me
1	Scottie	rap	KrJ	2012	89	{}	72180	en	en	en	My old girl left me on her old bull shit So I play it off like Pippen or
2	Pirate Password	rock	The never land pirate band	2011	175	{}	2122100	en	en	en	Avast there matey haha If a pirate asks ya for the password Yohoh
3	Indri	rock	Puta Volcano	2015	14	{}	6889288	en	en	en	Just throw a glimpse under the shell Ghostly voices shrill ecstatic
4	Maps	misc	ANBARDA	2018	4	{}	3735887	en	en	en	I miss the taste of a sweeter life I miss the conversation I'm search

Textlängen-Überblick (bereinigte Lyrics)

Durchschnittliche Länge: 1,565 Zeichen

Median: 1,218 Zeichen

Navigation

Ansicht auswählen:

- 1 Datensubset laden
- 2 Daten bereinigen
- 3 Tokenisierung
- 4 Statistische Analyse
- 5 Word Embedding
- 6 Model Evaluation
- 7 Text Classification
- 8 Text-Generierung

3 Kapitel 3 – Tokenization: Genius Song Lyrics Subset (1%)

Dataset: 34'049 Songs | 26'408 Artists | 6 Genres

Genres: Rap · Hip-Hop · Rock · Pop · R&B · Country · Miscellaneous

Purpose:

Vorbereitung der Textdaten für weitere Analysen, indem Songtexte in einzelne Tokens zerlegt und Stopwörter entfernt werden.

Diese Schritte sorgen für eine saubere und strukturierte Darstellung der Texte und bilden die Grundlage für nachgelagerte NLP- und Statistik-Analysen.

Hinweis: Dieser Abschnitt dokumentiert die Schritte aus dem zugehörigen Notebook `tokenization.ipynb`. Die Tokenisierung der Lyrics sowie das Entfernen von Stopwörtern wurden vollständig im Notebook ausgeführt. Die Streamlit-App lädt lediglich die dort erzeugten Daten und visualisiert ausgewählte Ergebnisse – ohne die Tokenisierung erneut durchzuführen.

1. Preparation

Es wird das im **Kapitel 2** bereinigte Datenset geladen, das bereits von Metadaten-Tags und Zeilenumbrüchen befreite Lyrics enthält.

```
df = pd.read_csv('data/clean/lyrics_subset_1pct_clean.csv')

df = df[df['language_cld3'] == "en"]
```

2. Tokenization

2.1 Build Tokens

Zuerst werden die Songtexte in einzelne Tokens zerlegt, d.h. jedes Wort wird aus dem vollständigen Text extrahiert.

Die resultierenden Tokens werden in einer neuen Spalte `words` gespeichert, zusätzlich wird `word_count` als Anzahl Tokens pro Song ergänzt.

```
def to_words(text: str) -> list[str]:
    if not isinstance(text, str):
        return []
    return text.split()
```

```
df["words"] = df["lyrics"].apply(to_words)
df["word_count"] = df["words"].apply(len)
```

2 Filter Stopwords

Anschließend werden Stopwörter entfernt.

Dadurch rücken inhaltlich bedeutungsvolle Wörter in den Vordergrund.

- Die gefilterten Tokens werden in `tokens` gespeichert
- Die Anzahl der Tokens pro Song in `token_count`

```
STOPWORDS = [
    "the", "a", "an", "and", "or", "but", "if", "then", "so", "than", "that", "those", "these", "this",
    "to", "of", "in", "on", "for", "with", "as", "at", "by", "from", "into", "over", "under", "up", "down",
    "is", "am", "are", "was", "were", "be", "been", "being", "do", "does", "did", "doing", "have", "has", "had",
    "i", "you", "he", "she", "it", "we", "they", "me", "him", "her", "us", "them", "my", "your", "his", "its", "our", "their",
    "not", "no", "yes", "yeah", "y'all", "yall", "im", "i'm", "i'd", "i'd", "i'll", "i'll", "you're", "you're", "don't", "don't",
    "cant", "can't", "ill", "i'll", "id", "i'd", "ive", "i've", "ya", "oh", "ooh", "la", "na", "nah"
]

def filtered_tokens(text):
    tokens = preprocess_text(text)
    return [t for t in tokens if t not in STOPWORDS and not t.isdigit() and len(t) > 1]

filtered_tokens("This is a test!")
```

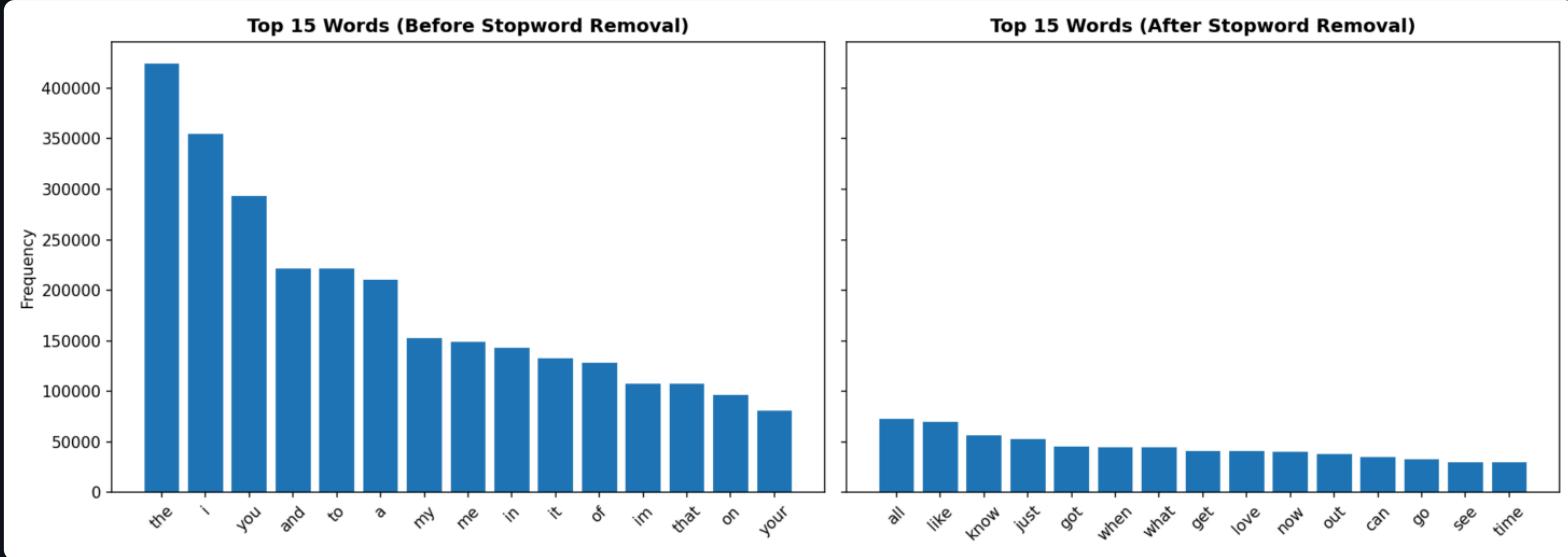
Tokenisierte Vorschau

	title	artist	words	word_count	tokes
0	2 Is Better 棍子	Chris Travis	Bitch I'm clean Two sticks like Chow Mein Two sticks li	294	bitch clean two sticks like chow mein
1	Scottie	KrJ	My old girl left me on her old bull shit So I play	199	old girl left old bull shit play off
2	Pirate Password	The never land pirate band	Avast there matey haha If a pirate asks ya for the pa	215	avast there matey haha pirate asks pa
3	Indri	Puta Volcano	Just throw a glimpse under the shell Ghostly voices shrill	162	just throw glimpse shell ghostly voices
4	Maps	ANBARDA	I miss the taste of a sweater life I miss the conversa	428	miss taste sweater life miss conversation

2.3 Visualisierung der häufigsten Wörter

Zur Veranschaulichung werden die häufigsten Wörter **vor** und **nach** dem Entfernen der Stopwörter gegenübergestellt.

Die Plots zeigen deutlich, dass das Entfernen von Stopwörtern die Verteilung stark verändert: Das häufigste Wort **nach** dem Filtern taucht in den ursprünglichen Top-15 gar nicht mehr auf.



3. Save final Dataset

Das finale Datenset enthält u.a. die neuen Spalten:

- `words` , `word_count`
- `tokens` , `token_count`

und dient als bereinigte und vorbereitete Basis für alle weiteren Textanalysen. Es wird unter `data/clean/data.csv` gespeichert.

3.1 Configuration

```
output_dir = "data/clean"
os.makedirs(output_dir, exist_ok=True)

output_path = os.path.join(output_dir, "data.csv")
```

Navigation

Ansicht auswählen:

- 1 Datensubset laden
- 2 Daten bereinigen
- 3 Tokenisierung
- 4 Statistische Analyse
- 5 Word Embedding
- 6 Model Evaluation
- 7 Text Classification
- 8 Text-Generierung

4 Kapitel 4 – Statistical Analysis: Genius Song Lyrics Subset (1%)

Dataset: 34'049 Songs | 26'408 Artists | 6 Genres

Genres: Rap / Hip-Hop · Rock · Pop · R&B · Country · Miscellaneous

Purpose:

Statistische Muster in den Songtexten untersuchen:

- Deskriptive Statistiken (Genre, Text-/Tokenlängen)
- Wort-Level-Analyse (Vokabular, Zipf's Law, Hapax Legomena)
- Category Statistics (pro Genre)
- N-Gramm-Analyse (Unigrams, Bigrams, Trigrams pro Dataset / Artist / Genre)

Der folgende Abschnitt dokumentiert das **Jupyter Notebook**.

Die eigentlichen Berechnungen und Plots laufen im Notebook und werden als PNG/JSON im Ordner `documentation/statistical_analysis` gespeichert.

Hinweis: Dieser Abschnitt dokumentiert die Schritte aus dem zugehörigen Notebook `statistical-analysis.ipynb`. Alle statistischen Auswertungen und Visualisierungen wurden vollständig im Notebook berechnet und als PNG/JSON gespeichert. Die Streamlit-App lädt diese Inhalte lediglich und zeigt sie an – ohne die Analysen erneut auszuführen.

1.1 Load Dataset

Laden des final bereinigten Datensatzes (`data/clean/data.csv`) und Rückkonvertierung der Spalten `words` und `tokens` von String-Repräsentationen zu echten Python-Listen (mittels `ast.literal_eval`).

```
df = pd.read_csv('data/clean/data.csv')

for col in ["words", "tokens"]:
    if isinstance(df[col].iloc[0], str):
        df[col] = df[col].apply(ast.literal_eval)
```

1.2 Descriptive Statistics

Zuerst wird die **Genre-Verteilung** analysiert und als Balkendiagramm geplottet. Anschliessend werden **Text- und Token-Statistiken** berechnet (total, min, avg, max) und jeweils als kleine Übersichtsgrafik gespeichert.

```
print("\nGENRE DISTRIBUTION")
print("=" * 60)
category_counts = df['tag'].value_counts().sort_values(ascending=False)

for tag, count in category_counts.items():
    pct = (count / len(df)) * 100
    print(f"{tag}: {count}, songs ({pct:.2f}%)")
```

2. Word-Level Analysis

2.1 Vocabulary Statistics

Bestimmung der Vokabulargrösse, Gesamtzahl der Worttokens und Type-Token Ratio (TTR). Nun werfen wir einen genaueren Blick auf die Texte und Wörter, indem wir das Vokabular analysieren, Zipfs Gesetz untersuchen, seltene Wörter (Hapaxlegomina) identifizieren und verschiedene Kategoriestatistiken untersuchen.

```
all_tokens = [token for tokens in df["words"] for token in tokens]

word_counts = Counter(all_tokens)
vocab_size = len(word_counts)
type_token_ratio = vocab_size / len(all_tokens)
```

VOCABULARY STATISTICS

```
=====
Total word tokens:          10,596,323
Unique words (vocabulary): 127,659
Type-token ratio:           0.0120
```

Im Durchschnitt kommt jedes Wort etwa 100 Mal im Datensatz vor, was auf einen hohen Wiederholungsgrad hindeutet. Das type-token ratio (TTR) von 0,012 ist relativ niedrig, was zu erwarten war, da der Korpus aus Songtexten besteht – einem Genre, das sich durch

wiederkehrende Wörter, Refrains und eine im Vergleich zu anderen Textarten begrenzte lexikalische Vielfalt auszeichnet.

2.2 Zipf-Analyse

Die Zipf-Law beschreibt eine fundamentale Eigenschaft natürlicher Sprache: Die Häufigkeit eines Wortes ist **umgekehrt proportional zu seinem Rang** in der sortierten Wortfrequenzliste.

Mathematische Form:

$$f(r) = \frac{C}{r^\alpha}$$

Bedeutung der Parameter:

- $f(r)$ = Häufigkeit des Wortes mit Rang r
- α = Exponent bzw. Steigung (typischer Idealwert für natürliche Sprache ≈ 1.0)
- C = Normierungskonstante

Wenn $\alpha = 1.0$, dann gilt:

- das Wort auf Rang 2 tritt **halb so häufig** auf wie das Wort auf Rang 1
- Rang 3 tritt **ein Drittel so häufig** auf
- Rang 4 **ein Viertel so häufig**, usw.

Diese Potenzgesetz-Struktur ist erstaunlich stabil und findet sich in unterschiedlichsten Texten, Genres, Korpora und Sprachen wieder.

```
all_word_freq = Counter(words).most_common(100)
ranks = list(range(1, len(all_word_freq) + 1))
```

```
frequencies = [freq for word, freq in all_word_freq]

log_ranks_100 = np.log(ranks).reshape(-1, 1)
log_freq_100 = np.log(frequencies)

model = LinearRegression()
model.fit(log_ranks_100, log_freq_100)

r_squared = model.score(log_ranks_100, log_freq_100)
slope = model.coef_[0]
intercept = model.intercept_
coefficient_C = np.exp(intercept)
```

2.3 Hapax Legomena (Rare Words)

Analyse der seltensten Wörter (Hapax Legomena, Count=1) und aller Wörter mit ≤ 5 Vorkommen.
Zusätzlich wird die Verteilung „Wie viele Wörter kommen X-mal vor?“ als Balkendiagramm gespeichert und Kennzahlen in `rare_words_stats.json` geschrieben.

```
word_counts = Counter(words)
hapax = [word for word, count in word_counts.items() if count == 1]
hapax_pct = (len(hapax) / vocab_size) * 100

rare_2 = [word for word, count in word_counts.items() if count == 2]
rare_3_5 = [word for word, count in word_counts.items() if 3 <= count <= 5]
rare_le_5 = len(hapax) + len(rare_2) + len(rare_3_5)
```

2.4 Category Statistics

Berechnung von Kennzahlen **pro Genre** (Tag):

- Anzahl Songs
- Gesamt- und Durchschnittswörter
- Vokabulargrösse
- Anteil Songs mit Zahlen im Text

Die Ergebnisse werden als Dreifach-Balkenplot gespeichert (`category_statistics.png`).

```
categories = df['tag'].unique()

category_stats = {}
for cat in categories:
    cat_df = df[df['tag'] == cat]
    cat_text = ' '.join(cat_df['lyrics'].str.lower())
    cat_words = cat_text.split()
    cat_vocab = len(set(cat_words))

    has_number_pct = sum(any(char.isdigit() for char in lyric) for lyric in

category_stats[cat] = {
    'songs': len(cat_df),
    'total_words': len(cat_words),
    'avg_words': len(cat_words) / len(cat_df),
    'vocab': cat_vocab,
    'has_number_pct': has_number_pct
}
```

3. N-gram Analysis

3.1 Unigram, Bigram, Trigram

Erstellung von Unigrams, Bigrams und Trigrams über alle Songs hinweg, Ausgabe der Top-15 pro N-Gramm-Typ und Speicherung als Plot `top15_ngrams.png`.

```
def ngrams(tokens, n):
    if n <= 0:
        return []
    iters = tee(tokens, n)
    for i, it in enumerate(iters):
        for _ in range(i):
            next(it, None)
    return zip(*iters)

    unigram_counts = Counter()
    bigram_counts = Counter()
    trigram_counts = Counter()

    for tokens in df["tokens"]:
        unigram_counts.update(tokens)
        bigram_counts.update(ngrams(tokens, 2))
        trigram_counts.update(ngrams(tokens, 3))

    top_unigrams = pd.DataFrame(unigram_counts.most_common(15), columns=["word",
    top_bigrams = pd.DataFrame([
        " ".join(k), v) for k, v in bigram_counts.most_common(15)],
```

```
        columns=["bigram", "count"]
    )
top_trigrams = pd.DataFrame(
    [(" ".join(k), v) for k, v in trigram_counts.most_common(15)],
    columns=["trigram", "count"]
)
```

3.2 N-Grams per Artist/Genre

Für jede Gruppe (Artist / Genre) wird das jeweils häufigste N-Gramm (Uni/ Bi/ Trigram) bestimmt und die Top-20 bzw. Top-Listen visualisiert.

Die Resultate werden als `top20_ngrams_per_artist.png` und `top_ngrams_per_genre.png` gespeichert.

```
def most_common_ngram_for_group(group_df: pd.DataFrame, label_col: str, n: int):
    """
    returns, for each group (artist/tag), the most frequent n-gram along with
    Columns: [label_col, 'ngram', 'count', 'songs']
    """
    rows = []
    for label, sub in group_df.groupby(label_col):
        c = Counter()
        for toks in sub["tokens"]:
            c.update(ngrams(toks, n))
        if c:
            top_ngram, cnt = c.most_common(1)[0]
            rows.append({label_col: label, "ngram": " ".join(top_ngram), "count": cnt})
        else:
```

```
rows.append({label_col: label, "ngram": None, "count": 0, "songs": songs})
return pd.DataFrame(rows).sort_values([label_col]).reset_index(drop=True)
```

Alle Resultate wurden im Notebook berechnet und als Grafiken gespeichert.

GENRE DISTRIBUTION TEXT STATISTICS TOKEN STATISTICS TOP 15 WORDS ZIPF'S LAW ANALYSIS RARE

ZIPF'S LAW ANALYSIS

Das angepasste Zipf-Law-Modell weist eine Steigung von $-0,83$ mit einem Wert von $0,98$ auf, was auf eine hervorragende Anpassung an die erwartete Verteilung hinweist. Obwohl die Steigung etwas flacher ist als die ideale $-1,0$, deutet diese geringe Abweichung ($0,17$) darauf hin, dass die Häufigkeits-Rang-Beziehung in den Liedtexten dem Zipf-law sehr nahe kommt – häufige Wörter werden viel häufiger verwendet als seltene, wie es typischerweise in Liedtexten zu beobachten ist.

ZIPF'S LAW ANALYSIS

Fitted equation: $f(r) = 788840.86 / r^{0.828}$

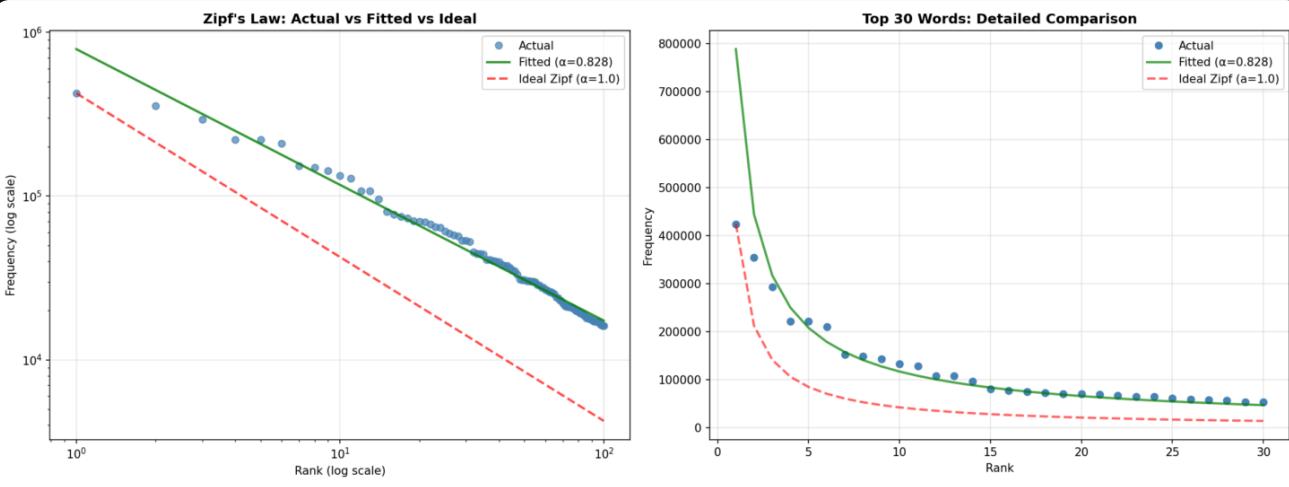
Model parameters:

Slope (α): -0.8284

R^2 (fit quality): 0.9845

Ideal Zipf slope: -1.0000

Deviation: 0.1716



Zipf Log-Log + Top-30 Comparison

Navigation

Ansicht auswählen:

- 1 Datensubset laden
- 2 Daten bereinigen
- 3 Tokenisierung
- 4 Statistische Analyse
- 5 Word Embedding
- 6 Model Evaluation
- 7 Text Classification
- 8 Text-Generierung

5 Kapitel 5 – Word Embedding: Genius Song Lyrics Subset (1%)

Dataset: 34'049 Songs | 26'408 Artists | 6 Genres
Genres: Rap / Hip-Hop · Rock · Pop · R&B · Country · Miscellaneous

Purpose:

Erstellung und Exploration von **Word Embeddings** für Songtexte.

Auf Basis der tokenisierten Lyrics werden:

- ein **Word2Vec-Modell** trainiert,
- semantische Beziehungen zwischen Wörtern untersucht,
- Songtexte über TF-IDF + Word2Vec auf **Dokument-Embeddings** abgebildet
- und diese im Embedding-Space analysiert,

Hinweis: Dieser Abschnitt dokumentiert die Schritte aus dem zugehörigen Notebook `word-embedding.ipynb`. Das Word2Vec-Modell und die Song-Embeddings wurden dort trainiert und als Dateien gespeichert. Die Streamlit-App lädt diese Inhalte lediglich und visualisiert die Ergebnisse – ohne das Modell erneut zu trainieren.

1. Train Word2Vec Model

Ziel: Lernen von Wortvektoren aus den Lyrics-Token mit `gensim.Word2Vec`.

Wichtige Parameter:

- `vector_size=50` → 50-dimensionale Embeddings (kompakt, schnell)
- `window=5` → Kontextfenster von 5 Wörtern links/rechts
- `min_count=2` → Wörter mit weniger als 2 Vorkommen werden ignoriert
- `epochs=100` → 100 Trainingsdurchläufe für stabilere Vektoren

```
model = Word2Vec(  
    sentences=sentences,  
    vector_size=50,  
    window=5,  
    min_count=2,  
    workers=4,  
    epochs=100  
)
```

	Parameter	Bedeutung
0	sentences	Liste von Wortlisten (Token pro Song)
1	vector_size	Dimension der Vektoren
2	window	Kontextfenstergroesse
3	min_count	Minimalhäufigkeit für Wörter
4	workers	Anzahl Threads
5	epochs	Trainingsdurchläufe

Die unteren Diagramme zeigen die Häufigkeit der 15 häufigsten Wörter vor und nach dem Entfernen von Stopwörtern. Wir können deutlich sehen, dass das Entfernen von Stopwörtern einen signifikanten Unterschied macht: Das häufigste Wort nach dem Filtern erscheint vor dem Entfernen der Stopwörter nicht einmal unter den 15 häufigsten Wörtern.

Ergebnis:

Ein trainiertes Word2Vec-Modell, das jedes Wort als Punkt in einem **50-dimensionalen Raum** repräsentiert – Wörter mit ähnlichem Kontext liegen nah beieinander.

3. TF-IDF & Dokument-Embeddings

Word2Vec liefert **Wortvektoren** – um ganze Songs zu repräsentieren, werden die Wortvektoren mit **TF-IDF gewichtet** gemittelt.

```
tfidf_vect = TfidfVectorizer(
    tokenizer=lambda x: x,
    preprocessor=lambda x: x,
    token_pattern=None,
    lowercase=False
)

X_tfidf = tfidf_vect.fit_transform(df["tokens"])
terms = tfidf_vect.get_feature_names_out()

dim = model.wv.vector_size
doc_emb_tfidf = np.zeros((X_tfidf.shape[0], dim), dtype=np.float32)

for i in range(X_tfidf.shape[0]):
    row = X_tfidf[i]
    if row.nnz == 0:
        continue
    idxs = row.indices
    wts = row.data
   vecs = []
    w = []
    for j, wt in zip(idxs, wts):
        vecs.append(model.wv.vectors[j])
        w.append(wt)
    doc_emb_tfidf[i] = np.array(vecs).mean(axis=0) * w / sum(w)
```

```

term = terms[j]
if term in model.wv:
    vecs.append(model.wv[term])
    w.append(wt)

if w:
    vecs = np.vstack(vecs)
    w = np.asarray(w, dtype=np.float32)
    doc_emb_tfidf[i] = (vecs * w[:, None]).sum(axis=0) / (w.sum() + 1e-9)

print("TF-IDF-Embeddings:", doc_emb_tfidf.shape)

keep = np.linalg.norm(doc_emb_tfidf, axis=1) > 0
df_use = df.reset_index(drop=True).loc[keep].reset_index(drop=True)
emb_use = doc_emb_tfidf[keep]
print("Nach Filter:", emb_use.shape)

```

Ergebnis: pro Song ein Embedding-Vektor, der beide Welten kombiniert:

- **Word2Vec** (semantische Struktur)
- **TF-IDF** (Gewichtung wichtiger Wörter)

4. Embedding of whole songs

Alternative: Song-Embeddings als einfacher Durchschnitt der Wortvektoren (`get_song_vector`), anschliessend Visualisierung im 3D-Raum und per PCA.

```

GENRE_COL = "tag"

def get_song_vector(tokens, w2v_model):
    """
        Compute a single vector representation for one song by
        averaging all word vectors for its tokens.
        If no token is in the vocabulary, return a zero vector.
    """
    if not isinstance(tokens, (list, tuple)):
        return np.zeros(w2v_model.vector_size, dtype=np.float32)

    vectors = [w2v_model.wv[t] for t in tokens if t in w2v_model.wv]

    if not vectors:
        return np.zeros(w2v_model.vector_size, dtype=np.float32)

    return np.mean(vectors, axis=0).astype(np.float32)

df_songs = df.dropna(subset=["tokens", GENRE_COL]).copy()

df_songs["embedding"] = df_songs["tokens"].apply(

```

```
        lambda toks: get_song_vector(toks, model)
    )

X = np.vstack(df_songs["embedding"].values)
y = df_songs[GENRE_COL].astype(str).values

print("Song embeddings shape:", X.shape)
print("Number of songs:", len(y))
print("Example genres:", y[:10])
```

5. Save Model & Features

Zum Schluss werden alle wichtigen Artefakte für die App und spätere Modelle gespeichert:

- `data/features/song_embeddings.npy` – Song-Embedding-Matrix `X`
- `data/features/song_labels.npy` – Genre-Labels `y`
- `data/features/song_metadata.csv` – Metadaten (Genre, Titel, Artist,...)
- `models/word2vec_lyrics.model` – trainiertes Word2Vec-Modell

```
os.makedirs("data/features", exist_ok=True)

np.save("data/features/song_embeddings.npy", X)
np.save("data/features/song_labels.npy", y)

print("Saved song embeddings and labels to 'data/features/'")

meta_cols = [GENRE_COL]
for col in ["title", "artist", "id", "song_id"]:
    if col in df_songs.columns:
        meta_cols.append(col)

df_songs[meta_cols].to_csv("data/features/song_metadata.csv", index=False)
print("Saved song metadata to 'data/features/song_metadata.csv'")

os.makedirs("models", exist_ok=True)
model.save("models/word2vec_lyrics.model")
print("Saved Word2Vec model to 'models/word2vec_lyrics.model'")
```



Notebook-Resultate – Word Embeddings



Word2Vec-Modell & ähnliche Wörter

- Vocabulary size: 65,504

- Vector size: 50

Wort für Ähnlichkeits-Suche:

love

Top 10 ähnliche Wörter zu **love** :

	word	similarity
0	loving	0.71
1	baby	0.70
2	babe	0.70
3	everything	0.70
4	cause	0.70
5	true	0.69
6	lov	0.69
7	darlin	0.68
8	know	0.68
9	girl	0.68

📦 Song-Embeddings (Dokument-Vektoren)

- Embedding-Matrix X: 34,049 Songs × 50 Dimensionen
- Anzahl Labels: 34,049

📋 Beispiel-Metadaten

	tag	title	artist	id
0	rap	2 Is Better 棍子	Chris Travis	30362
1	rap	Scottie	KrJ	721
2	rock	Pirate Password	The never land pirate band	21223
3	rock	Indri	Puta Volcano	68892
4	misc	Maps	ANBARDA	37350

⚠ PCA 2D – Song-Embedding Space (Ausschnitt)

Songs im Embedding-Space (PCA 2D, Ausschnitt)



Navigation

Ansicht auswählen:

- 1 Datensubset laden
- 2 Daten bereinigen
- 3 Tokenisierung
- 4 Statistische Analyse
- 5 Word Embedding
- 6 Model Evaluation
- 7 Text Classification
- 8 Text-Generierung

6 Kapitel 6 - Model Evaluation: Genius Song Lyrics Subset (1%)

Dataset: 34'049 Songs | 26'408 Artists | 6 Genres

Genres: Rap · Hip-Hop · Rock · Pop · R&B · Country · Miscellaneous

Purpose:

Mehrere Modelle zur automatischen Genre-Klassifikation vergleichen – basierend auf unterschiedlichen Textrepräsentationen (Embeddings) und Klassifikatoren.

Embeddings:

- Word2Vec (self-trained)
- TF-IDF (character-level n-grams)
- SentenceTransformer (MiniLM)

Classifier:

- LinearSVC
- Logistic Regression
- Random Forest

Ausgewertet werden:

- Accuracy & Balanced Accuracy
- F1-Macro
- Klassifikationsberichte (im Notebook)
- Normalisierte Confusion Matrices (als PNG gespeichert)

Hinweis: Dieser Abschnitt dokumentiert die Schritte aus dem zugehörigen Notebook `model-evaluation.ipynb`. Das Training der Modelle sowie die Berechnung aller Metriken und Confusion Matrices wurden vollständig im Notebook durchgeführt und als Ergebnisse gespeichert. Die Streamlit-App lädt diese Ergebnisse ausschließlich und visualisiert sie – ohne die Modelle erneut zu trainieren.

1. Preparation

1.1 Load Dataset

Laden des final bereinigten Datensatzes (`data/clean/data.csv`) und Konvertierung der Spalte `tokens` von einer String-Repräsentation in echte Python-Listen (mittels `ast.literal_eval`).

Anschließend erfolgt das **Label-Encoding**: Die Genre-Bezeichnungen (Strings wie "rap", "rock", "rb") werden mit einem `LabelEncoder` in ganze Zahlen umgewandelt, da Klassifikationsmodelle numerische Labels benötigen.

```
df = pd.read_csv("data/clean/data.csv")

df["tokens"] = df["tokens"].apply(ast.literal_eval)
texts = df["tokens"]
labels = df["tag"]

label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(labels)
```

1.2 Train-Test-Split

Der Datensatz wird in einen **Trainings-** und einen **Testsplit** aufgeteilt. Dabei werden 80 % der Daten zum Trainieren der Modelle verwendet, die restlichen 20 % dienen zur unabhängigen Evaluation.

Durch `stratify=y_encoded` wird sichergestellt, dass alle Genres im gleichen Verhältnis in beiden Splits vertreten sind – wichtig bei **unausgeglichenen Klassen**.

```
X_train_texts, X_test_texts, y_train, y_test = train_test_split(
    texts,
    y_encoded,
    test_size=0.2,
    random_state=42,
    stratify=y_encoded,
)
```

2. Embeddings & Modelle

2.1 Word2Vec

2.1.1 Embedding erzeugen

Für die erste Embedding-Strategie wird ein **Word2Vec-Modell** auf den Token-Sequenzen des Trainingssplits trainiert. Word2Vec lernt für jedes Wort einen dichten Vektor, der semantische Ähnlichkeiten abbildet (z. B. ähnliche Wörter → ähnliche Vektoren).

Um jedes Dokument (Songtext) als festen Embedding-Vektor darzustellen, werden die Wortvektoren gemittelt (**Mean Word Embedding**).

Dies erzeugt einen robusten, kompakten Repräsentationsvektor pro Song.

```
w2v = Word2Vec(
    sentences=X_train_tokens,
    vector_size=100,
    window=5,
    min_count=5,
    workers=4,
    sg=1,
    epochs=10,
    seed=42,
)

def embed_sentence(tokens, model):
    vectors = [model.wv[w] for w in tokens if w in model.wv]
    if len(vectors) == 0:
        return np.zeros(model.vector_size)
    return np.mean(vectors, axis=0)

X_train_w2v = np.vstack([embed_sentence(toks, w2v) for toks in X_train_tokens])
X_test_w2v = np.vstack([embed_sentence(toks, w2v) for toks in X_test_tokens])
```

2.1.2 Klassifikation auf Word2Vec

Auf Basis der erzeugten Word2Vec-Embeddings werden drei unterschiedliche Klassifikationsmodelle trainiert.

Alle Modelle erhalten `class_weight="balanced"`, um die ungleich verteilten Genres auszugleichen und Minderheitsklassen nicht zu benachteiligen.

Im Folgenden werden die drei Modelle jeweils einzeln gezeigt.

`LinearSVC` ist ein lineares SVM-Modell und eignet sich gut für hochdimensionale Text-Embeddings.

```
clf_w2v_svc = LinearSVC(class_weight="balanced", max_iter=10000)
clf_w2v_svc.fit(X_train_w2v, y_train)
y_pred_w2v_svc = clf_w2v_svc.predict(X_test_w2v)
```

Die `Logistische Regression` ist ein einfaches, stabiles lineares Modell und funktioniert gut bei unausgeglichenen Klassen.

```
clf_w2v_logreg = LogisticRegression(
    max_iter=2000,
    n_jobs=-1,
    class_weight="balanced",
)
clf_w2v_logreg.fit(X_train_w2v, y_train)
y_pred_w2v_logreg = clf_w2v_logreg.predict(X_test_w2v)
```

Der `Random Forest` ist ein nichtlineares Ensemblemodell. Er kann komplexe Muster erfassen, skaliert aber weniger gut mit hochdimensionalen Text-Embeddings.

```
clf_w2v_rf = RandomForestClassifier(  
    n_estimators=400,  
    max_depth=20,  
    min_samples_leaf=3,  
    max_features="sqrt",  
    class_weight="balanced",  
    n_jobs=-1,  
    random_state=42,  
)  
clf_w2v_rf.fit(X_train_w2v, y_train)  
y_pred_w2v_rf = clf_w2v_rf.predict(X_test_w2v)
```

2.2 TF-IDF

2.2.1 Embedding erzeugen

Für die zweite Embedding-Strategie wird **TF-IDF** auf Zeichen-n-Grammen angewendet.

Die Token-Sequenzen werden dazu wieder zu Strings zusammengefügt, anschließend wird ein TF-IDF-Vektorraum auf **Character n-grams (3–5)** gelernt.

Damit lassen sich charakteristische Schreibweisen, Silbenmuster und typische Endungen pro Genre erfassen.

```
X_train_texts_char = X_train_texts.apply(lambda toks: " ".join(toks))  
X_test_texts_char = X_test_texts.apply(lambda toks: " ".join(toks))  
  
tfidf = TfidfVectorizer(  
    analyzer="char",  
    ngram_range=(3, 5),  
    min_df=5,  
    max_df=0.9,  
)  
  
X_train_tfidf = tfidf.fit_transform(X_train_texts_char)  
X_test_tfidf = tfidf.transform(X_test_texts_char)
```

2.2.2 Klassifikation auf TF-IDF

Auf den TF-IDF-Features werden erneut drei Klassifikationsmodelle trainiert: **LinearSVC**, **Logistic Regression** und **Random Forest**, jeweils mit `class_weight="balanced"`.

LinearSVC eignet sich auch hier gut für die hochdimensionalen TF-IDF-Vektoren.

```
clf_tfidf_svc = LinearSVC(class_weight="balanced")  
clf_tfidf_svc.fit(X_train_tfidf, y_train)
```

```
y_pred_tfidf_svc = clf_tfidf_svc.predict(X_test_tfidf)  
Die Logistische Regression dient als weiteres lineares Basismodell auf TF-IDF.
```

```
clf_tfidf_logreg = LogisticRegression(  
    max_iter=2000,  
    n_jobs=-1,  
    class_weight="balanced",  
)  
clf_tfidf_logreg.fit(X_train_tfidf, y_train)  
y_pred_tfidf_logreg = clf_tfidf_logreg.predict(X_test_tfidf)
```

Der Random Forest bildet die nichtlineare Vergleichsbasis auf TF-IDF-Features.

```
clf_tfidf_rf = RandomForestClassifier(  
    n_estimators=400,  
    max_depth=20,  
    min_samples_leaf=3,  
    max_features="sqrt",  
    class_weight="balanced",  
    n_jobs=-1,  
    random_state=42,  
)  
clf_tfidf_rf.fit(X_train_tfidf, y_train)  
y_pred_tfidf_rf = clf_tfidf_rf.predict(X_test_tfidf)
```

2.3 Transformer (SentenceTransformer MiniLM)

2.3.1 Embedding erzeugen

Als dritte Embedding-Strategie wird ein SentenceTransformer verwendet: `all-MiniLM-L6-v2` erzeugt semantische Satz- bzw. Dokument-Embeddings direkt aus den vollständigen Songtexten.

Dazu werden die Token-Sequenzen wieder zu Strings zusammengefügt und mit dem SentenceTransformer zu dichten Vektoren encodiert.

```
model = SentenceTransformer("all-MiniLM-L6-v2", device="cpu")  
  
X_train_sent = [" ".join(toks) for toks in X_train_texts]  
X_test_sent = [" ".join(toks) for toks in X_test_texts]  
  
X_train_emb_st = model.encode(  
    X_train_sent,  
    batch_size=16,  
    show_progress_bar=True,
```

```
        convert_to_numpy=False,
        convert_to_tensor=True,
    )

X_test_emb_st = model.encode(
    X_test_sent,
    batch_size=16,
    show_progress_bar=True,
    convert_to_numpy=False,
    convert_to_tensor=True,
)

# Tensor → Python-Listen (für die Sklearn-Modelle)
X_train_emb_st = X_train_emb_st.tolist()
X_test_emb_st = X_test_emb_st.tolist()
```

2.3.2 Klassifikation auf Transformer-Embeddings

Auf den Transformer-Embeddings werden erneut drei Klassifikationsmodelle trainiert: **LinearSVC**, **Logistic Regression** und **Random Forest**.

LinearSVC dient hier als robustes lineares Modell auf den semantischen Embeddings.

```
clf_st_svc = LinearSVC(class_weight="balanced", max_iter=10000)
clf_st_svc.fit(X_train_emb_st, y_train)
y_pred_st_svc = clf_st_svc.predict(X_test_emb_st)
```

Die **Logistische Regression** wird als zweites lineares Vergleichsmodell verwendet.

```
clf_st_logreg = LogisticRegression(
    max_iter=2000,
    n_jobs=-1,
    class_weight="balanced",
)
clf_st_logreg.fit(X_train_emb_st, y_train)
y_pred_st_logreg = clf_st_logreg.predict(X_test_emb_st)
```

Für den **Random Forest** werden die Embeddings in NumPy-Arrays konvertiert.

```
X_train_st_rf = np.asarray(X_train_emb_st)
X_test_st_rf = np.asarray(X_test_emb_st)

clf_st_rf = RandomForestClassifier(
    n_estimators=400,
    max_depth=20,
    min_samples_leaf=3,
```

```
        max_features="sqrt",
        class_weight="balanced",
        n_jobs=-1,
        random_state=42,
    )

clf_st_rf.fit(X_train_st_rf, y_train)
y_pred_st_rf = clf_st_rf.predict(X_test_st_rf)
```

3. Speichern des finalen Modells & der Evaluationsergebnisse

```
os.makedirs("models", exist_ok=True)
joblib.dump(clf_st_svc, "models/clf_st_svc.joblib")
joblib.dump(label_encoder, "models/label_encoder.joblib")

MODELS_DIR = Path("models")
MODELS_DIR.mkdir(exist_ok=True)

eval_file = MODELS_DIR / "eval_results.json"
eval_file.write_text(json.dumps(results, indent=2), encoding="utf-8")

cm_path = MODELS_DIR / "confusion_matrix_best.npy"
np.save(cm_path, cm_best)
```



Notebook-Resultate – Word Embeddings

Übersicht über alle Modelle

	model	embedding	classifier	accuracy	balanced_accuracy	f1_macro
0	tfidf_LogReg	TF-IDF (char 3-5 n-grams)	LogisticRegression	0.551	0.535	0.4
1	tfidf_LinearSVC	TF-IDF (char 3-5 n-grams)	LinearSVC	0.593	0.458	0.
2	w2v_LinearSVC	Word2Vec	LinearSVC	0.576	0.510	0.
3	st_LinearSVC	SentenceTransformer(all-MiniLM-L6-v2)	LinearSVC	0.572	0.515	0.
4	tfidf_RandomForest	TF-IDF (char 3-5 n-grams)	RandomForest	0.581	0.405	0.
5	w2v_RandomForest	Word2Vec	RandomForest	0.646	0.403	0.
6	st_LogReg	SentenceTransformer(all-MiniLM-L6-v2)	LogisticRegression	0.475	0.545	0.
7	w2v_LogReg	Word2Vec	LogisticRegression	0.461	0.549	0.
8	st_RandomForest	SentenceTransformer(all-MiniLM-L6-v2)	RandomForest	0.624	0.343	0.

F1-Macro nach Modell



Details zu den Modellen (inkl. Confusion Matrices)

Word2Vec – LinearSVC Word2Vec – Logistic Regression Word2Vec – Random Forest TF-IDF – LinearSVC TF-IDF – Logistic Regression TF-IDF – Random Forest Transformer (MiniLM) – LinearSVC

Accuracy

Balanced Accuracy

F1-Macro

0.572

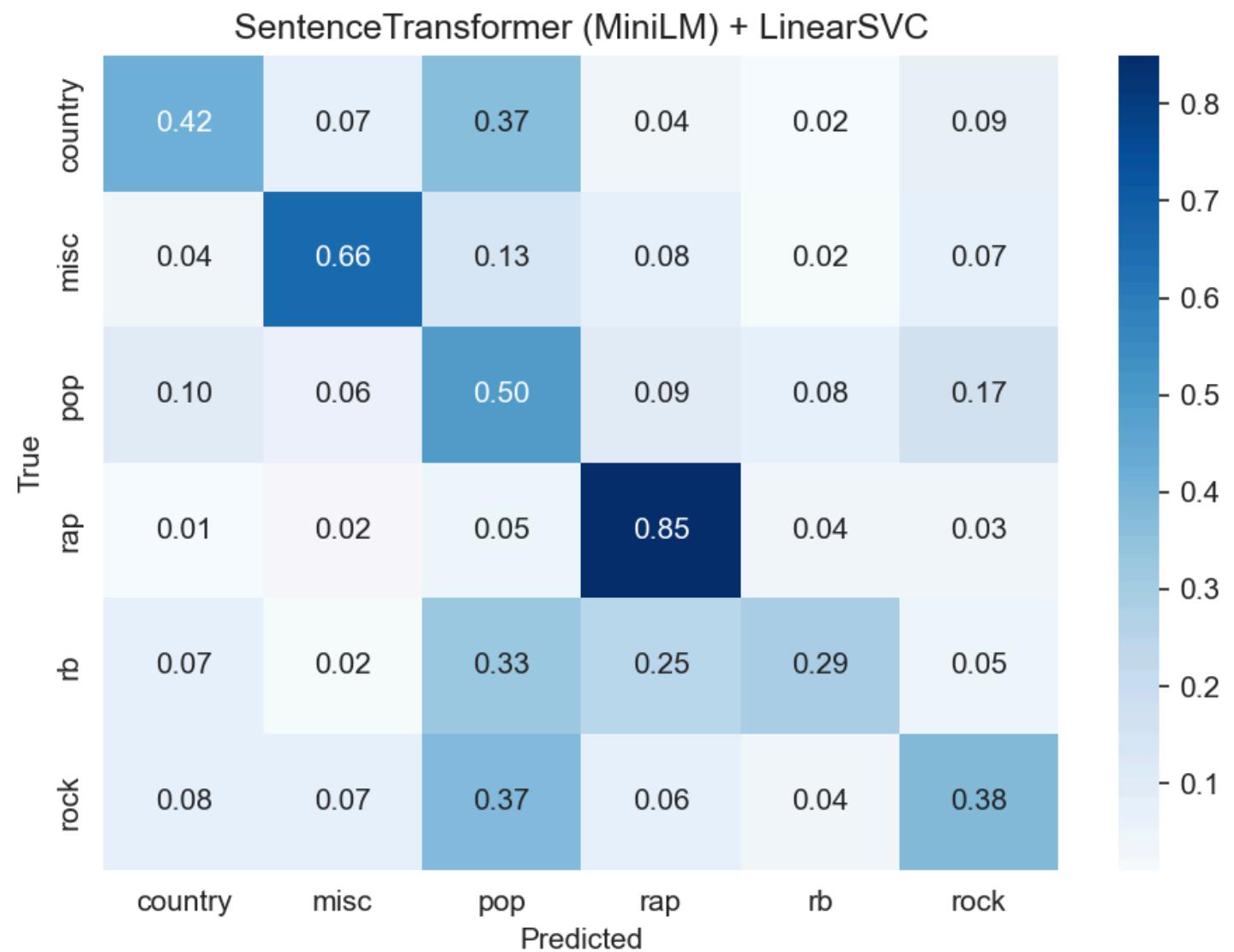
0.515

0.452

Embedding: SentenceTransformer(all-MiniLM-L6-v2)

Classifier: LinearSVC

Confusion Matrix



Transformer (MiniLM) – Zusammenfassung der Klassifikatoren

LinearSVC

- Accuracy: ~0.572
- Balanced Accuracy: ~0.515
Beste Balance zwischen Genauigkeit und Fairness.

Logistische Regression

- Accuracy: ~0.475
- Balanced Accuracy: ~0.543
Fairste und ausgewogenste Klassifikation.

Random Forest

- Accuracy: ~0.624
- Balanced Accuracy: ~0.343
Sehr hohe Accuracy, aber extrem geringe Fairness.

Fazit

- Beste Gesamtperformance: **LinearSVC**
 - Beste Fairness: **Logistische Regression**
 - Höchste Accuracy, aber schlechteste Fairness: **Random Forest**
-

Finale Modellwahl & Modellselektion

Über alle drei Embedding-Strategien – **Word2Vec**, **TF-IDF** und **Transformer (MiniLM)** – zeigt sich ein konsistentes Muster:

- **LinearSVC** liefert die stabilste Gesamtperformance, unabhängig vom Embedding.
- **Logistische Regression** verbessert systematisch die Klassenbalance und den Recall für Minderheitsgenres.
- **Random Forest** erreicht oft hohe Accuracy, ist aber deutlich zugunsten der Mehrheitsklassen verzerrt und erzielt eine niedrige Balanced Accuracy.

🎯 Final gewähltes Modell

SentenceTransformer (MiniLM) + LinearSVC

Dieses Modell bietet:

- solide Accuracy (~0.57)
- die beste Balanced Accuracy unter den leistungsstarken Modellen (~0.52)
- gute Performance sowohl für dominante als auch für Minderheitsgenres
- robuste Generalisierung dank semantisch reichhaltiger Transformer-Embeddings

In Kombination mit **LinearSVC**, das sehr stabil auf hochdimensionalen Embeddings arbeitet, ergibt sich ein Modell, das eine gute Balance zwischen Performance und Fairness über alle Genres hinweg bietet.

Navigation

Ansicht auswählen:

- 1 Datensubset laden
- 2 Daten bereinigen
- 3 Tokenisierung
- 4 Statistische Analyse
- 5 Word Embedding
- 6 Model Evaluation
- 7 Text Classification
- 8 Text-Generierung

7 Kapitel 7 - Text Classification: Genius Song Lyrics (1%)

Dataset: 34'049 Songs | 26'408 Artists | 6 Genres

Genres: Rap / Hip-Hop · Rock · Pop · R&B · Country · Miscellaneous

Purpose:

Verwendung des im Notebook `model-evaluation.ipynb` gewählten **besten Modells**, um neue Songtexte automatisch einem Genre zuzuordnen.

Dieses Kapitel dient als **Prototyp** für eine interaktive Text-Classification-Demo:

- Klassifikation **einzelner Lyrics**
- Klassifikation **mehrerer Lyrics (Batch)**

Ausgewähltes Modell:

SentenceTransformer (all-MiniLM-L6-v2) + LinearSVC

Hinweis: Dieser Abschnitt basiert auf dem zugehörigen Notebook `text-classification.ipynb`. Das dort geladene und vorbereitete Modell wird in der Streamlit-App lediglich angewendet, um neue Lyrics zu klassifizieren – ohne erneutes Training.

1. Load Trained Model and Label Encoder

Laden des im Kapitel *Model Evaluation* gewählten **finalen Klassifikationsmodells** (SentenceTransformer + LinearSVC) sowie des `LabelEncoder` für die Genres.

```
clf_st_svc = joblib.load("models/clf_st_svc.joblib")
label_encoder = joblib.load("models/label_encoder.joblib")

st_model = SentenceTransformer("all-MiniLM-L6-v2", device="cpu")
```

2. Classification

2.1 Classification of one Lyric

Beispiel: Ein einzelner Songtext wird gereinigt, mit MiniLM eingebettet und über den LinearSVC klassifiziert.

```
lyrics = """
Yeah I'm driving through the city late at night,
lights low, bass loud, trouble on my mind...
"""

lyrics_clean = lyrics.strip()

embedding_tensor = st_model.encode(
    [lyrics_clean],
    batch_size=16,
    show_progress_bar=False,
    convert_to_numpy=False,
    convert_to_tensor=True,
```

```
)  
  
embedding = embedding_tensor.tolist()  
  
pred_idx = clf_st_svc.predict(embedding)[0]  
pred_genre = label_encoder.inverse_transform([pred_idx])[0]
```

2.2 Classification of more Lyrics

Im zweiten Schritt werden mehrere kurze Beispiel-Lyrics in einem Rutsch klassifiziert, um das Modellverhalten zu demonstrieren.

```
texts = [  
    "Yeah, I'm riding through the city with my homies late at night...",  
    "Baby, I miss you every single day, I can't get you off my mind...",  
    "Whiskey on the dashboard, small town lights and dusty roads...",  
    "The crowd is roaring, the drums are loud, the stage is burning..."  
]  
  
emb = st_model.encode(  
    [t.strip() for t in texts],  
    convert_to_numpy=False,  
    convert_to_tensor=True,  
    show_progress_bar=False,  
)  
emb_list = emb.tolist()  
  
pred_idx = clf_st_svc.predict(emb_list)  
pred_genres = label_encoder.inverse_transform(pred_idx)
```

🔮 Interaktive Demo – Genre-Vorhersage für neue Lyrics

SentenceTransformer + LinearSVC erfolgreich geladen.

Genres: country, misc, pop, rap, rb, rock

2.1 Einzelnen Songtext klassifizieren

Gib hier deinen Songtext ein:

Yeah I'm driving through the city late at night, lights low, bass loud, trouble on my mind...



Genre vorhersagen

Vorhergesagtes Genre: **rock**

2.2 Mehrere Lyrics auf einmal klassifizieren

Gib mehrere Songtexte ein, **einer pro Zeile**.

Kurze Fragmente reichen bereits, das Modell arbeitet mit Kontextsignalen.

Mehrere Lyrics (eine Zeile = ein Text):

```
Yeah, I'm riding through the city with my homies late at night...
Baby, I miss you every single day, I can't get you off my mind...
Whiskey on the dashboard, small town lights and dusty roads...
The crowd is roaring, the drums are loud, the stage is burning...
```



Alle Zeilen klassifizieren

2.3 Interpretation

Die im Notebook gezeigten Vorhersagen wirken **intuitiv**:

- „*City + homies + late night*“ → **Rock**
(könnte auch Rap sein, aber die Stimmung ist eher „rebellisch/rockig“)
- „*I miss you every single day*“ → **Country**
(klassisches Heartbreak-Thema)
- „*Whiskey + dusty roads + small town*“ → eindeutig **Country**
- „*Crowd, drums, stage is burning*“ → **Pop**
(klare Stadion-/Performance-Energie)

Fazit:

Das Modell weist Genres auf Basis kurzer Texte den typischen lyrischen Themen sehr plausibel zu. Selbst knappe Ausschnitte reichen, um stilistische Hinweise sinnvoll zu nutzen.

Navigation

Ansicht auswählen:

- 1 Datensubset laden
- 2 Daten bereinigen
- 3 Tokenisierung
- 4 Statistische Analyse
- 5 Word Embedding
- 6 Model Evaluation
- 7 Text Classification
- 8 Text-Generierung

8 Kapitel 8 - Lyrics Generation: Genius Song Lyrics (1%)

Dataset: 34'049 Songs | 26'408 Artists | 6 Genres

Genres: Rap / Hip-Hop · Rock · Pop · R&B · Country · Miscellaneous

Purpose:

Generierung neuer, stilkonsistenter Songtexte mithilfe eines einfachen **Markov-Chain-Modells**, das auf den bestehenden Lyrics trainiert wird.

Unterstützte Optionen:

- Generierung aus dem **kompletten Datensatz**
- **Genre-spezifische** Generierung

Hinweis: Dieser Abschnitt dokumentiert die Schritte aus dem zugehörigen Notebook `text-generation.ipynb`. Die grundlegende Vorgehensweise zur Markov-basierten Lyrics-Generierung wurde dort entwickelt. Die Streamlit-App übernimmt diese Logik und ermöglicht eine interaktive Generierung neuer Songzeilen und Songs.

1. Imports and Setup

1.1 Import Libraries and Load Data

Zunächst werden `pandas` zum Laden des Datensatzes und `markovify` für das Markov-Chain-Modell importiert.

Anschließend wird der bereinigte Datensatz geladen und ein erster Blick auf die relevanten Spalten (`lyrics`, `tag`) geworfen.

```
df = pd.read_csv("data/clean/data.csv")  
  
df[["lyrics", "tag"]].head()
```

1.2 Data Preparation

Alle vorhandenen Lyrics werden gesammelt und zu einem großen Text-Korpus zusammengefügt, der als Trainingsbasis für das Markov-Modell dient.

```
all_lyrics = df["lyrics"].dropna().tolist()  
  
corpus_text = "\n".join(all_lyrics)
```

2. Markov Chain Model

2.1 Kurze Einordnung: Markov-Ketten

Markov-Modelle arbeiten nur mit lokalen Übergangswahrscheinlichkeiten: Die nächste Zeile hängt also immer nur vom aktuellen Zustand bzw. den letzten Wörtern ab. Trotz dieser Einfachheit entstehen oft stilistische Muster, die an die Original-Lyrics erinnern.

Für wirklich kohärente, inhaltlich konsistente Songs wären allerdings komplexere neuronale Sprachmodelle nötig. In diesem Kapitel geht es bewusst um eine leichtgewichtige, gut erklärbare Demo.

2.2 Build Model

Aus dem gesamten Textkorpus wird ein **Markov-Chain-Modell** mit `state_size=2` gebaut.

Zur Generierung einzelner Zeilen wird

`model.make_short_sentence(max_chars=90, tries=100)` verwendet:

- `make_short_sentence` erzeugt einen **gültigen Satz** mit maximal `max_chars` Zeichen.
- Besser geeignet für kurze, lyrics-ähnliche Zeilen als `make_sentence()`.
- `tries=100` steuert, wie viele Versuche unternommen werden, bevor aufgegeben wird.

So bleiben die generierten Zeilen **knapp** und erinnern an typische Songtext-Zeilen.

```
text_model_all = markovify.Text(corpus_text, state_size=2)
```

2.3 Generate a few lines

Generiert einige Beispiel-Zeilen aus dem **gesamten Korpus**.

```
print("== Generated lyrics (full corpus) ==\n")
for _ in range(10):
    line = text_model_all.make_short_sentence(max_chars=90, tries=100)
    if line:
        print(line)
```

2.4 Genre-specific Lyrics

Für eine **genre-spezifische** Generierung wird zunächst ein Subset der Lyrics nach Tag (`df["tag"]`) gefiltert und daraus ein neues Markov-Modell gebaut. Die Funktion `generate_markov_lyrics` kapselt dieses Verhalten.

```
def generate_markov_lyrics(genre=None, num_lines=10):
    """Generate Markov-based lyrics from the full corpus or a specific genre
    if genre is None:
        subset = df["lyrics"].dropna().tolist()
        label = "full corpus"
    else:
        subset = df[df["tag"] == genre]["lyrics"].dropna().tolist()
        label = f"genre: {genre}"

    corpus_text = "\n".join(subset)
    model = markovify.Text(corpus_text, state_size=2)

    print(f"== Generated lyrics ({label}) ==\n")
    for _ in range(num_lines):
        line = model.make_short_sentence(max_chars=90, tries=100)
        if line:
            print(line)
```

```
generate_markov_lyrics(genre="country", num_lines=10)
```

2.5 Lyrics with Verse and Chorus

Für komplexere Songstrukturen werden Hilfsfunktionen definiert:

- `generate_line` – eine einzelne Zeile mit Markovify
- `generate_verse` – mehrere Zeilen als **Strophe**
- `generate_chorus` – **Refrain** mit teilweise wiederkehrenden Zeilen
- `generate_song` – baut einen Song aus `[Verse 1]`, `[Chorus]`, `[Verse 2]`

```
def generate_line(model, max_tries=100):  
    line = model.make_short_sentence(max_chars=90, tries=100)  
    return line if line else ""  
  
def generate_verse(model, num_lines=8):  
    lines = []  
    for _ in range(num_lines):  
        line = generate_line(model)  
        if line:  
            lines.append(line)  
    return lines  
  
def generate_chorus(model, num_lines=4):  
    lines = []  
    base_line = generate_line(model)  
    if not base_line:  
        base_line = "La la la"
```

```
for i in range(num_lines):
    if i % 2 == 0:
        lines.append(base_line)
    else:
        line = generate_line(model)
        lines.append(line if line else base_line)
return lines

def generate_song(model):
    verse1 = generate_verse(model)
    chorus = generate_chorus(model)
    verse2 = generate_verse(model)

    print("[Verse 1]")
    print("\n".join(verse1))
    print("\n[Chorus]")
    print("\n".join(chorus))
    print("\n[Verse 2]")
    print("\n".join(verse2))

genre = "country"
subset = df[df["tag"] == genre]["lyrics"].dropna().tolist()
text_model_genre = markovify.Text("\n".join(subset), state_size=2)

generate_song(text_model_genre)
```



Interaktive Lyrics-Generierung

1) Einstellungen

Genre für das Markov-Modell:

country

Anzahl Zeilen für einfache Generierung:

8

2) Einfache Zeilen generieren

Zeilen generieren

3) Song mit Vers & Chorus generieren

Zeilen pro Vers:

4

Zeilen im Chorus:

3

Song generieren (Verse + Chorus)

Generated song (genre: country)

[Verse 1]

Yellow roses Are you the black sheep?
If you didn't then I heard the news They laid him in a Wrangler with the tide!
Cause I don't think you are, a super S-T-A-R?
Forever enshrined behind some glass pane In an old tired out rodeo man Who do you mean?

[Chorus]

Runnin round in this world of pain: Lord, Lord, live brains!
This all started when I lie How do I need?
Runnin round in this world of pain: Lord, Lord, live brains!

[Verse 2]

Yet we have you and me Just as long as we have no fear When I'm faced with the tide!
I want to live?
Oh yeah, I gotta be country music, you can kiss my oh my gah!
.A Cheaters game just breaking me down Why'd you have to love people?