Task 3: Evaluierung

In folgendem Abschnitt werden unsere Recommender hergestellt und getestet. Alle erstellten Funktionen und verwendete Bibliotheken befinden sich in unerer Helper-Datei 'p.py'. Dies erleichtert uns den Überblick über das gesammte Notebook.

In [ ]:
```
# import functions and libraries from helper-file p
from p import *
```

Daten einlesen & bereinigen

Wir lesen hier die Dateien ein und führen sie zusammen anhand der tmdbId --> So bekommen wir die entsprechende movieIds

In [ ]:
```
# read data
df_links = pd.read_csv("ml-25m/links.csv")
df_tmdb_movies = pd.read_csv("tmdb_movies.csv", sep="\t")

# rename id column so that we can merge later
df_tmdb_movies.rename(columns={'id':'tmdbId'}, inplace=True)

# merge movielens movies with tmdb movies
df_movies = pd.merge(df_links, df_tmdb_movies, on='tmdbId')
df_movies.head()
```

Out[ ]:

	movieId	imdbId	tmdbId	Unnamed: 0	adult	backdrop_path	belongs_to
0	895	105729	79782.0	0	False	/s2bpgVhpWODDfoADW78IpMDCMTR.jpg	
1	895	105729	79782.0	13349	False	/s2bpgVhpWODDfoADW78IpMDCMTR.jpg	
2	181393	1684935	79782.0	0	False	/s2bpgVhpWODDfoADW78IpMDCMTR.jpg	
3	181393	1684935	79782.0	13349	False	/s2bpgVhpWODDfoADW78IpMDCMTR.jpg	
4	1115	114472	141210.0	1	False		NaN

5 rows × 28 columns

Hier extrahieren wir die relevanten Daten aus den angegebenen Spalten, weil sie dort verschachtelt sind.

In [ ]:
```
# extract data from dictionaries and list, separate key values by '|'
## extract function
def extract_values(data):
    return data.apply(lambda x: "|".join([i["name"] for i in eval(x)]))

## actual extraction
df_movies["genres"] = extract_values(df_movies["genres"])
df_movies["spoken_languages"] = extract_values(df_movies["spoken_languages"])
df_movies["production_companies"] = extract_values(df_movies["production_companies"]
df_movies["production_countries"] = extract_values(df_movies["production_countries"]
```

Hier haben wir die Spalten definiert und geordnet, die wir benötigen (alle anderen wurden automatisch entfernt)

In [ ]:
```
col_order = [
    "movieId",
    "title",
    "genres",
    "overview",
    "release_date",
    "runtime",
    "original_language",
    "spoken_languages",
    "production_companies",
    "production_countries",
]
# reorder columns
df_movies = df_movies[col_order]
```

Wir entfernen Filme mit einer Laufzeit von unter 30 Minuten und über 300 Minuten, da wir für unseren Recommender keine Kurz - oder Langfilme beachten wollen.

In [ ]:
```
# drop rows with runtime under 30 --> remove short films because they are not releva
df_movies = df_movies.drop(df_movies[df_movies["runtime"] < 30].index)

# drop rows with runtime over 300 --> remove long films because they are not relevan
df_movies = df_movies.drop(df_movies[df_movies["runtime"] > 300].index)
```

Hier wurden die Daten weiter bereinigt; MovieId Duplikate und leere Filmbeschreibungen entfernt

In [ ]:
```
# remove duplicate rows (there are some duplicate movies with different movieId's, t
df_movies = df_movies.drop_duplicates(subset = df_movies.columns.difference(["movieI

# remove all overviews that are nan
df_movies = df_movies.dropna(subset=["overview"])
```

Jetzt exportieren wir den bereinigten Dataframe, damit wir die enthaltenen Daten später bei der explorativen Datenanalyse untersuchen können.

In [ ]:
```
# export df_movies to csv
df_movies.to_csv('ml-25m/movies_clean.csv', index = False)
```

Movie Ratings

Hier lesen wir die Filmbewertungen ein.

So können wir dann Bewertungen plausibilisieren --> falls diese Daten unmöglich sind (zb eine Bewertung wurde vor der Veröffentlichung vom Film erstellt), werden sie gelöscht.

In [ ]:
```
df_ratings = pd.read_csv("ml-25m/ratings.csv")
```

Wir haben Filme von der TMDB API abgefragt und gespeichert (df_movies) und wir wollen jetzt nur die Bewertungen behalten, deren Filme dich in df_movies befinden. Darum schauen wir, ob die movields übereinstimmen, falls ja dann werden diese Bewertungen beibehalten. Alle andere Bewertungen werden automatisch gelöscht, da sie irrelevant sind.

In [ ]:
```
# only keep movie ratings where the movieId is in df_movies
df_ratings = df_ratings[df_ratings["movieId"].isin(list(df_movies["movieId"]))]
```

Danach konvertieren wir die "timestamp" Spalte zu einem "datetime" Objekt und speichern das in einer neuen Spalte namens "date". Die Spalte "timestamp" wird anschliessend gelöscht.

In [ ]:
```
# convert timestamp to datetime
df_ratings["date"] = [datetime.fromtimestamp(x) for x in df_ratings["timestamp"]]

# drop the old timestamp column
df_ratings.drop("timestamp", axis = 1, inplace = True)
```

Jetzt werden beide Dataframes zusammengeführt.

In [ ]:
```
# merge movies and ratings on movieId
df_movies_ratings = pd.merge(df_ratings, df_movies, on = "movieId")
```

Wir müssen jetzt die Daten plausibilisieren.

Wir setzen dies um, indem wir Bewertungen löschen, welche vor dem Erscheinungsdatum abgegeben wurden.

In [ ]:
```
# drop rows where the ratings were submitted before the movie released
df_movies_ratings = df_movies_ratings[df_movies_ratings["date"] > df_movies_ratings[

# drop columns because they are useless now
df_movies_ratings.drop(["date", "release_date"], axis = 1, inplace = True)
```

Hier ist ein weiteres Beispiel von Daten zu plausibilisieren; Wir schauen, ob es Users gibt, welche den gleichen Film mehrmals bewertet haben.

In [ ]:
```
df_movies_ratings[df_movies_ratings.duplicated(subset = ["userId", "movieId"])]
```

Out[ ]:

	userId	movieId	rating	title	genres	overview	runtime	original_language	spoken_languages	pr

Export vom Dataframe für seperates Explorative Datenanalyse Python Notebook.

In [ ]:
```
df_movies_ratings.to_csv('ml-25m/df_movies_ratings.csv', index=False)
```

NLP Implementierung

Bei unserer Challenge haben wir uns für folgende drei nlp-Algortihmen entschieden:

· TF-IDF (Termhäufigkeit - Inverse Dokumentenhäufigkeit) ist ein numerisches Mass, das verwendet wird, um zu reflektieren, wie wichtig ein Wort für ein Dokument in einer Sammlung oder einem Korpus ist. Die Wichtigkeit steigt proportional zur Anzahl der Zeiten, die das Wort im Dokument erscheint, wird jedoch durch die Häufigkeit des Wortes im Korpus ausgeglichen. Dies hilft, die Tatsache auszugleichen, dass einige Wörter im Allgemeinen häufiger vorkommen.

· Die Latente Semantische Analyse (LSA) ist eine Technik, die verwendet wird, um Beziehungen zwischen einem Satz von Dokumenten und den von ihnen enthaltenen Begriffen zu analysieren, indem sie eine Menge von Konzepten erstellt, die mit den Dokumenten und Begriffen verbunden sind. LSA repräsentiert Dokumente und Begriffe in einem niedrigdimensionalen Raum und identifiziert Beziehungen zwischen ihnen aufgrund der Nähe der Dokumente und Begriffe in diesem Raum.

· Doc2vec ist eine Methode zur Darstellung von Dokumenten als Vektoren in einem hochdimensionalen Raum. Es ähnelt word2vec, das Wörter als Vektoren darstellt, aber doc2vec berücksichtigt den Kontext, in dem Wörter in einem Dokument erscheinen. Doc2vec kann verwendet werden, um Dokument-Embeddings zu erzeugen, die die Bedeutung eines Dokuments erfassen, die in verschiedenen Aufgaben der natürlichen Sprachverarbeitung verwendet werden können.

Anschliessend werden wir alle 3 Algorithmen miteinander vergleichen und uns für einen entscheiden für den Prototyp.

Pre-Processing: Stemming

Bevor man mit einem NLP Algorithmus beginnt, müssen die Daten zuerst darauf vorbereitet werden (pre-processed)

Dafür haben wir in der Datei p.py eine Funktion "stemming_tokenizer" erstellt, die verschiedene Varianten eines Wortes auf ihren gemeinsamen Wortstamm zurückführt, z.B. Swimming --> Swim.

Die Funktion löscht zudem auch noch unnötige Symbole wie Kommas und Punkte und auch "stop words". Das sind z.B. Wörter wie "the", "a" ect.

Wichtig zu wissen ist, dass die "pre-processed" Daten für alle 3 NLP Algorithmen gleich verwendet werden können.

In [ ]:
```
# copy dataframe to avoid overwriting the original one
df_movies_nlp = df_movies.copy().reset_index(drop = True)
```

```
# loop through the chosen columns and stem the texts
for col in ["overview", "spoken_languages", "original_language", "production_compani
    df_movies_nlp[col] = stemming_tokenizer(df_movies_nlp[col])

# create new title column with stemmed title (we don't want to stem the original tit
df_movies_nlp["new_title"] = stemming_tokenizer(df_movies_nlp["title"])
```

Hier sehen wir, dass wir die ausgewählten Texte auf ihren Wortstamm zurückgeführt haben.

In [ ]:
```
df_movies_nlp.head(3)
```

Out[ ]:

	movieId	title	genres	overview	release_date	runtime	original_language	sp
0	895	Venice	Drama|Romance	atmospher come age stori featur imagin young b...	2010-06-11	110	pl	
1	2679	A Place at the Table	Documentary	use person stori power documentari illumin pli...	2012-03-22	84	en	
2	4249	Kingdom Come	Comedy|Documentary	documentari kingdom come follow first time dir...	2011-01-01	88	en	

Kombinationsentscheid von Spalten für den perfekten Score

Wir schreiben eine Funktion namens 'create_combinations', die in der Datei 'p.py' enthalten ist. Die Funktion fügt zufällig die angegebene Anzahl von Spalten hinzu, um systematisch herauszufinden, welche Kombination von Spalten (Daten) die höchsten Scores oder die besten Vorschläge für Filme hat.

Folgende Attribute wollen wir für die Berechnung verwenden

- Titel
- Overview
- Original Language
- Spoken Language
- Production Companies
- Production Countries

Wir rufen die Funktion 'create_combinations' auf und sie erstellt einen Dataframe, der alle Filme mit 10 Filmvorschlägen enthält. Danach wird der Dataframe als CSV-Datei exportiert und wieder eingelesen, damit wir die Funktion nicht jedes Mal ausführen müssen, wenn wir die Ergebnisse ansehen wollen.

In [ ]:
```
# get recommendations for each movie based on the combinations
df_recommendations = recommendations_per_comb(df_movies_nlp, df_movies)

# sort the recommendations by score1
df_recommendations.sort_values(by = ["movie", "score_1"], ascending = False)
```

```
# export dataframe to csv for later
df_recommendations.to_csv("rec.csv", index = False)
```

Hier lesen wir die Datei ein und sehen, dass für das Dataframe eine Dimension von 378880 x 22 hat (~ 19000 Filme * 20 Kombinationen). Also es wurden 10 ähnliche Filme pro Film pro Spaltenkombination vorgeschlagen.

In [ ]:
```
# read the recommendations dataframe
rec = pd.read_csv("ml-25m/rec.csv", index_col = "Unnamed: 0")

# show dimension
print(rec.shape)

rec.head()
```

```
(378880, 22)
```

Out[ ]:

	combination	movie	recommended_movie_1	score_1	recommended_movie_2	score_2
0	['new_title', 'overview', 'spoken_languages', ...	Venice	Panic Attack	0.154583	7 Things You Don't Know About Men	0.148710
1	['new_title', 'overview', 'spoken_languages', ...	A Place at the Table	Food and Shelter	0.229952	Food Stamped	0.085962
2	['new_title', 'overview', 'spoken_languages', ...	Kingdom Come	Chronic-Con, Episode 420: A New Dope	0.069982	Kingdom Come	0.069428
3	['new_title', 'overview', 'spoken_languages', ...	Camille Claudel, 1915	Rodin	0.173721	Ducoboo 2: Crazy Vacation	0.124234
4	['new_title', 'overview', 'spoken_languages', ...	My Kingdom	White Vengeance	0.098080	The Final Master	0.080325

5 rows × 22 columns

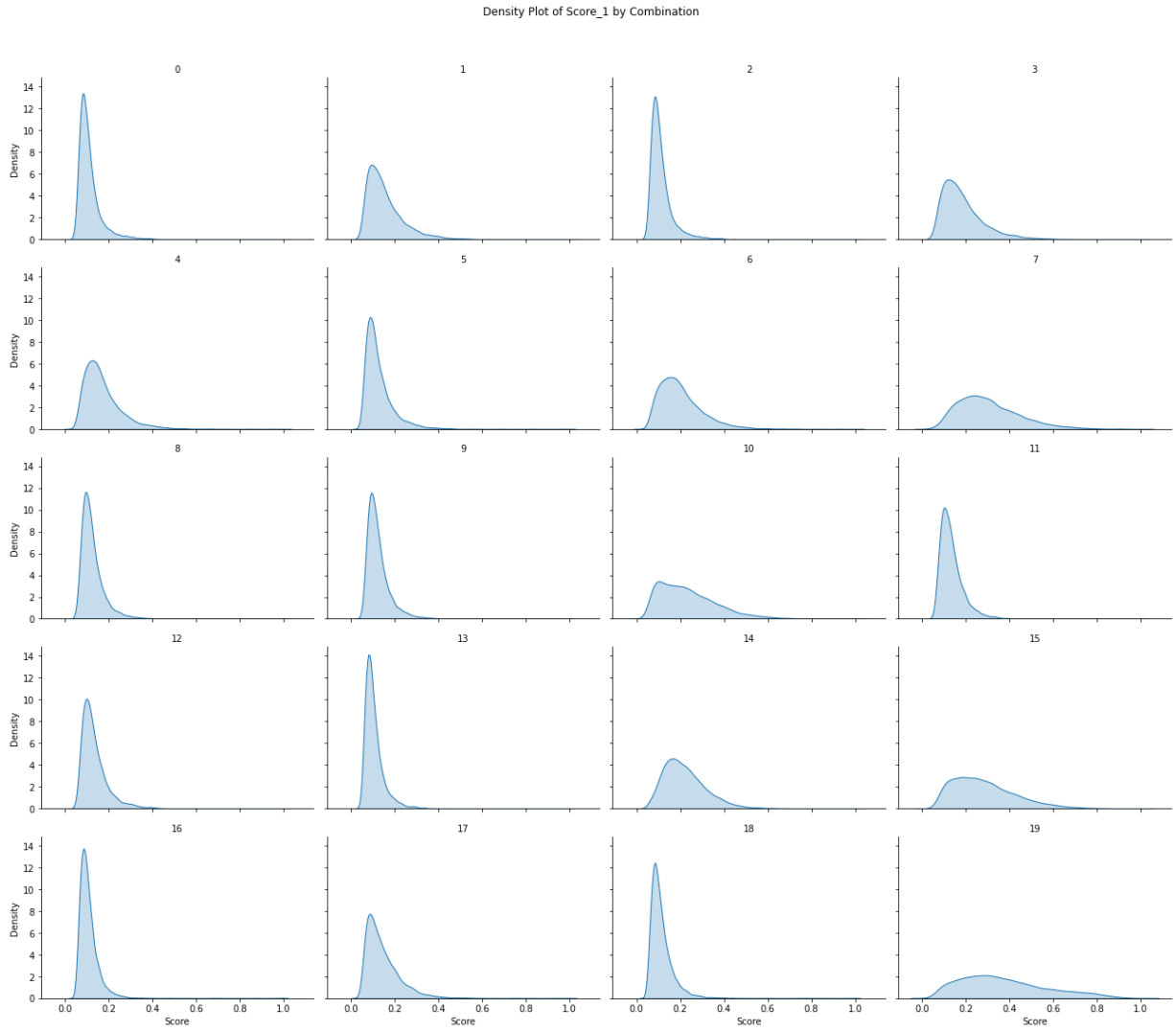
Wir erstellen hier eine neue Spalte, um die unique Kombinationen indexieren zu können.

In [ ]:
```
rec["set_combination"] = rec.groupby("combination").ngroup()
rec["set_combination"].head()
```

Out[ ]:
```
0    9
1    9
2    9
3    9
4    9
Name: set_combination, dtype: int64
```

Von allen Kombinationen wollen wir die Score-Verteilung vom Recommended Movie 1 anschauen. Dafür erstellen wir einen Facet Plot. Die Nummern pberhalb der kleinen Plots entspricht die Kombinationsnummer (0-19).

In [ ]:
```
g = sns.FacetGrid(rec, col = "set_combination", col_wrap = 4, height = 3, aspect = 1
g.map(sns.kdeplot, "score_1", fill = True)
g.set_titles(col_template = "{col_name}")
g.set_axis_labels("Score", "Density")
g.fig.suptitle("Density Plot of Score_1 by Combination", y = 1.05)
plt.show()
```



Kombination Entscheidung

Wir entscheiden uns für die Kombination 19, weil die Verteilung die höchsten Werte annimmt.

Overview kommt 2 Mal vor, das bedeutet, dass die Wörter im Overview höcher bewertet werden (das Ziel ist es ja eher anhand des Overviews Filme vorzuschlagen)

Die Kombination sieht wie folgt aus:

In [ ]:
```
rec[rec["set_combination"] == 19]["combination"].unique()
```

Out[ ]:
```
array(["['new_title', 'overview', 'spoken_languages', 'original_language', 'producti
on_companies', 'production_countries', 'new_title', 'new_title', 'overview', 'spoken
_languages', 'spoken_languages', 'spoken_languages', 'spoken_languages', 'original_l
anguage', 'original_language', 'original_language', 'original_language', 'original_l
anguage', 'original_language', 'original_language', 'production_companies', 'product
ion_companies', 'production_companies', 'production_companies', 'production_companie
s', 'production_companies', 'production_companies', 'production_companies', 'product
ion_companies', 'production_companies', 'production_countries', 'production_countrie
```

```
s', 'production_countries', 'production_countries', 'production_countries', 'product
ion_countries', 'production_countries', 'production_countries']"],
      dtype=object)
```

In [ ]:
```
# cols from combination 19
cols = ['new_title', 'overview', 'spoken_languages', 'original_language', 'productio
```

Kombination Testen

Wir testen die Kombination nun auf Filme, die wir als Gruppe sehr gut kennen: "Fast & Furious 6", "Interstellar", "Toy Story 3"

Dabei ist für uns erkennbar, dass die Filmvorschläge recht gut sind.

In [ ]:
```
# we can use head(1) because the combination 19 is the first row of the dataframe
rec[rec["movie"] == "Fast & Furious 6"].sort_values(by = "score_1", ascending = Fals
```

Out[ ]:

	combination	movie	recommended_movie_1	score_1	recommended_movie_2	score_
20788	['new_title', 'overview', 'spoken_languages', ...	Fast & Furious 6	Furious 7	0.452835	The Fate of the Furious	0.44282

1 rows × 23 columns

In [ ]:
```
rec[rec["movie"] == "Interstellar"].sort_values(by = "score_1", ascending = False).h
```

Out[ ]:

	combination	movie	recommended_movie_1	score_1	recommended_movie_2	score
21458	['new_title', 'overview', 'spoken_languages', ...	Interstellar	Inception	0.36404	Dunkirk	0.272

1 rows × 23 columns

In [ ]:
```
rec[rec["movie"] == "Toy Story 3"].sort_values(by = "score_1", ascending = False).he
```

Out[ ]:

	combination	movie	recommended_movie_1	score_1	recommended_movie_2	score_2
19039	['new_title', 'overview', 'spoken_languages', ...	Toy Story 3	Incredibles 2	0.774336	Toy Story 4	0.771716

1 rows × 23 columns

TF-IDF

Nun testen wir den TF-IDF Algorithmus basierend auf 2 Filme "21 Jump Street" & "Interstellar"

In [ ]:
```
recommendations_tfidf = tfidf_final(["21 Jump Street", "Interstellar"], df_movies_nl
print(recommendations_tfidf)
```

```
['Battle: Los Angeles', '22 Jump Street', 'Skyfall', 'Inception', 'Dunkirk']
```

LSA

LSA (Latent Semantic Analysis) auch bekannt als LSI (Latent Semantic Index) verwendet ein- Bag-of-Word-Modell (BoW), das zu einer Term-Dokument-Matrix führt (Vorkommen von Begriffen in einem Dokument)

Die Zeilen stehen für Begriffe und die Spalten für Dokumente. LSA lernt latente Themen, indem eine Matrixzerlegung der Dokument-Term-Matrix unter Verwendung der Singulärwertzerlegung durchgeführt wird. LSA Wird in der Regel zur Dimensionsreduzierung eingesetzt.

Um unser Algorithmus kurz zu testen wollen wir 5 Filme vorschlagen, basierend auf beide Filme "Fast Five"

In [ ]:
```
recommendations_lsa = lsa_final(["Fast Five"], df_movies_nlp, cols)
print(recommendations_lsa)
```

```
['The Fate of the Furious', 'Furious 7', 'Snow White and the Huntsman', 'Honey: Rise
Up and Dance', 'American Reunion']
```

Doc2Vec

Doc2Vec ist ein Algorithmus, der verwendet wird, um numerische Repräsentationen von Dokumenten zu erstellen. Es ist eine Erweiterung von Word2Vec, einem Algorithmus, der ähnlich funktioniert, aber stattdessen numerische Repräsentationen von Wörtern lernt. Doc2Vec nutzt das Konzept von "Document-Embeddings", die den Kontext und die Bedeutung eines Dokuments in einem Vektor darstellen. Doc2Vec lernt diese Embeddings in unbeaufsichtigter Weise, indem es ein neuronales Netzwerk auf einem grossen Korpus von Dokumenten trainiert. Es ist ein mächtiges Werkzeug für die Verarbeitung von Textdaten und hat sich in vielen Anwendungen bewährt.

Parameter für das doc2vec Modell:

- vector_size: Eine grössere Vektorgrösse erfasst mehr Informationen über die Filme, erfordert jedoch auch mehr Rechenleistung und ist anfälliger für Überanpassung. Um für uns die passendste Vektorgrösse zu wählen haben wir mit den Vektorgrössen experimentiert, sprich wir haben viele Modelle erstellt, welche eine andere Vektorgrösse besitzt. Schlussendlich haben wir uns für 90 entschieden.
- window: Eine grössere Fenstergrösse erfasst mehr Kontext zu den Filmen. Wie bei der Vektorgrösse haben wir mit verschiedenen windows experimentiert und entschieden uns für 5.
- min_count: Wenn man einen grossen Datensatz mit vielen seltenen Wörtern verfügt, kann das Festlegen einer höheren min_count dazu beitragen, die Grösse des Modells zu reduzieren und die Leistung zu verbessern. Da wir einen recht grossen Datensatz haben, welche recht viele seltene Wörter hat, haben wir uns für eine min_count von 2 entschieden.
- workers: in Doc2Vec bezieht sich auf die Anzahl der Worker-Threads, die beim Trainieren des Modells verwendet werden sollen. Dieser Parameter kann verwendet werden, um den

Trainingsprozess zu beschleunigen, indem die Berechnung über mehrere CPU-Kerne hinweg parallelisiert wird.

In [ ]:
```
# recommend top 5 movies based on the the following movies: The Amazing Spider-Man a
recommendations_d2v = doc2vec_final(df_movies_nlp, ["The Amazing Spider-Man", "Venom
print(recommendations_d2v)
```

```
['Inequality for All' 'Retake' 'Juggernaut'
 'Spider-Man: Into the Spider-Verse' 'Uncharted']
```

Qualitative Plausibilisierung

Eine qualitative Plausibilisierung bezieht sich auf die Überprüfung, ob die Ergebnisse eines Movie Recommenders aus Sicht des Benutzers sinnvoll und nachvollziehbar sind. Dies umfasst die Überprüfung, ob die empfohlenen Filme dem Benutzer gefallen könnten und ob sie sinnvoll in Bezug auf die bisherigen Filmauswahlen des Benutzers sind.

Eine qualitative Plausibilisierung ist wichtig, um sicherzustellen, dass der Movie Recommender dem Benutzer tatsächlich nützliche Empfehlungen liefern kann. Sie hilft auch dabei, eventuelle Fehler oder Ungereimtheiten in der Empfehlungslogik zu erkennen und zu beheben.

Um diese qualitative Plausibilisierung durchzuführen, haben wir 5 Personen ausgewählt, welche unseren Recommender benutzen können. Bei den 5 ausgewählten Personen haben wir darauf geachtet, dass das Alter, das Geschlecht und der Filmgeschmack unterschiedlich sind.

Person 1

Person 1 (23, männlich) ist ein Pokemon Fan und möchte die Vorschläge für den Film "Pokémon the Movie: The Power of Us" (Genre: Animation, Family, Adventure, Fantasy) wissen.

TFIDF

In [ ]:
```
recommendations_tfidf = tfidf_final(["Pokémon the Movie: The Power of Us"], df_movie

for movie in recommendations_tfidf:
    print(movie)
```

```
Pokémon the Movie: I Choose You!
Pokémon Detective Pikachu
Pokémon the Movie: Kyurem vs. the Sword of Justice
Pokémon the Movie: Hoopa and the Clash of Ages
Pokémon the Movie: Diancie and the Cocoon of Destruction
```

LSA

In [ ]:
```
recommendations_lsa = lsa_final(["Pokémon the Movie: The Power of Us"], df_movies_nl

for movie in recommendations_lsa:
    print(movie)
```

```
Isoroku Yamamoto, the Commander-in-Chief of the Combined Fleet
Gintama: The Movie: The Final Chapter: Be Forever Yorozuya
Pokémon the Movie: I Choose You!
Naruto Shippuden the Movie: Blood Prison
Fusé: Memoirs of a Huntress
```

Doc2Vec

In [ ]:
```
recommendations_doc2vec = doc2vec_final(df_movies_nlp, ["Pokémon the Movie: The Powe

for movie in recommendations_doc2vec:
    print(movie)
```

```
Hercules
Around June
Patema Inverted
The Runner from Ravenshead
Petals on the Wind
```

Bemerkung Person 1

Unserer Person 1 gefallen grundsätzlich die meisten vorgeschlagenen Filme, aber die Filme, welche mit dem TF-IDF Algorithmus vorgeschlagen wurden, findet er am besten.

Person 2

Person 2 (22, weiblich) ist ein Fan von deutschen romantischen Komödien und möchte die Vorschläge für den Film "What a Man" (Genre: Comedy, Romance) wissen.

TFIDF

In [ ]:
```
recommendations_tfidf = tfidf_final(["What a Man"], df_movies_nlp, cols)

for movie in recommendations_tfidf:
    print(movie)
```

```
Walk Like a Panther
The Pasta Detectives
The Most Beautiful Day
Break Up Man
Playing Doctor
```

LSA

In [ ]:
```
recommendations_lsa = lsa_final(["What a Man"], df_movies_nlp, cols)

for movie in recommendations_lsa:
    print(movie)
```

```
Playing Doctor
The Pasta Detectives
Bloodbrotherz
Break Up Man
Hitman: Agent 47
```

Doc2Vec

In [ ]:
```
recommendations_doc2vec = doc2vec_final(df_movies_nlp, ["What a Man"], cols)

for movie in recommendations_doc2vec:
    print(movie)
```

```
4:44 Last Day on Earth
The Woman Who Dreamed of a Man
```

```
Boy Meets Girl
The Death and Life of Otto Bloom
Celal and Ceren
```

Bemerkung Person 2

Unserer Person 2 gefallen mit Abstand die 5 vorgeschlagenen Filme vom TF-IDF Algorithmus am besten. Bei den anderen beiden Algorithmen gab es viele Filme, welche zu einem komplett anderen Genre gehören.

Person 3

Person 3 (34, männlich) ist ein Fan von Actionfilme und möchte die Vorschläge für den Film "Green Zone" (Genre: War, Action, Adventure, Drama, Thriller) wissen.

TFIDF

In [ ]:
```
recommendations_tfidf = tfidf_final(["Green Zone"], df_movies_nlp, cols)

for movie in recommendations_tfidf:
    print(movie)
```

```
Johnny English Reborn
About Time
Paul
Les Misérables
Senna
```

LSA

In [ ]:
```
recommendations_lsa = lsa_final(["Green Zone"], df_movies_nlp, cols)

for movie in recommendations_lsa:
    print(movie)
```

```
Johnny English Reborn
Contraband
Senna
Johnny English Strikes Again
Les Misérables
```

Doc2Vec

In [ ]:
```
recommendations_doc2vec = doc2vec_final(df_movies_nlp, ["Green Zone"], cols)

for movie in recommendations_doc2vec:
    print(movie)
```

```
The Culture High
Warcraft
Extinction
The Zohar Secret
Finding Altamira
```

Bemerkung Person 3

Unserer Person 3 überzeugen die unterschiedlichen Algorithmen nicht wirklich. Es gibt bei allen 3 Gruppen Filme, welche die ihm gefallen und auch nicht gefallen, weil sie teilweise auch

komplett andere Genres haben.

Person 4

Person 4 (46, weiblich) sieht eher selten Filme und falls doch, schaut sie sich gerne Dokumentationen an. Deshalb möchte sie Vorschläge für den Dokumentarfilm "The Last Mountain" (Genre: Documentary) wissen.

TFIDF

In [ ]:
```
recommendations_tfidf = tfidf_final(["The Last Mountain"], df_movies_nlp, cols)

for movie in recommendations_tfidf:
    print(movie)
```

```
Greater
Political Animals
The Lovers
Old Man
The Last Fall
```

LSA

In [ ]:
```
recommendations_lsa = lsa_final(["The Last Mountain"], df_movies_nlp, cols)

for movie in recommendations_lsa:
    print(movie)
```

```
Enduring Destiny
Pee-wee's Big Holiday
Cash Only
Answer This!
Bear Nation
```

Doc2Vec

In [ ]:
```
recommendations_doc2vec = doc2vec_final(df_movies_nlp, ["The Last Mountain"], cols)

for movie in recommendations_doc2vec:
    print(movie)
```

```
Hot Coffee
Rise of the Planet of the Apes
009 Re:Cyborg
Sol Invictus
Generation Iron
```

Bemerkung Person 4

Auch wenn nicht alle vorgeschlagenen Filme von der Person 4 gesehen wurden, würde sie eher die Filme anschauen von den Algorithmen TF-IDF und doc2vec. Bei LSA wurden zu viele Filme von anderen Genres vorgeschlagen.

Person 5

Person 5 (28, männlich) sieht abends gerne Horrorfilme und möchte deshalb die Vorschläge für den Film "It" (Genre: Horror, Fantasy) wissen.

TFIDF

In [ ]:
```
recommendations_tfidf = tfidf_final(["It"], df_movies_nlp, cols)

for movie in recommendations_tfidf:
    print(movie)
```

```
Death Note
It Chapter Two
New Year's Eve
The Lego Ninjago Movie
Wolves at the Door
```

LSA

In [ ]:
```
recommendations_lsa = lsa_final(["It"], df_movies_nlp, cols)

for movie in recommendations_lsa:
    print(movie)
```

```
Fist Fight
It Chapter Two
Horrible Bosses 2
Horrible Bosses
Going the Distance
```

Doc2Vec

In [ ]:
```
recommendations_doc2vec = doc2vec_final(df_movies_nlp, ["It"], cols)

for movie in recommendations_doc2vec:
    print(movie)
```

```
Six Degrees of Celebration 2
The Scapegoat
Dracula Untold
Wet Bum
Brigsby Bear
```

Bemerkung Person 5

Unserer Person 5 gefallen hauptsächlich die vorgeschlagenen Filme vom Algorithmus TF-IDF. Auch wenn der Lego Film und die Komödie New Year's Eve vorgeschalgen werden.

Fazit der qualitativen Plausibilisierung

Bei den Genres Animation, Family, Adventure, Fantasy haben die drei nlp-Algorithmen sehr gut perform, vor allem TF-IDF. Leider performten sie bei den Genres Comedy und Romance nicht so gut. Der einzige Algorithmus, der einigermassen gute Filmvorschläge bei ihnen gab, war TF-IDF. Auch bei den Genres War, Action, Drama und Thriller gab es leider keine guten Filmvorschläge. Dafür funktionierte der Recommender mit dem TF-IDF und doc2vec Algorithmus gut bei dem Genre Documentary. Beim Genre Horror brachte auch nur der TF-IDF gute Fimvorschläge.

Grundsätzlich sind wir zufrieden mit dem Ergebnis dieser Evaluierung. Leider funktionierte der Recommender nur bei gewissen Genres mit allen drei nlp-Algorithmen gut. Um das Problem bei diesen Genres zu lösen, würden wir das Genre beim Modellieren hinzufügen. Leider haben wir

aber zu wenig Zeit, um unsere Problemlösung zu testen. Deswegen haben wir uns entschieden für alle Genres den TF-IDF Algorithmus zu nutzen. Über alle Genres hinweg, performte dieser Algorithmus am besten.

Train Test Split

Wir wenden jetzt die Train-Test-Split Methode an, um die Leistung unserer Modelle zu bewerten. Wir teilen dabei die Daten in 2 Datensätze auf: Training & Test. Den Trainingssatz verwenden wir, um das Modell zu trainieren, während der Testsatz dazu verwendet wird, die Leistung des Modells anhand unsichtbarer Daten zu bewerten. Dies ist wichtig, da wir wissen möchten, wie gut mein Modell mit neuen, zuvor nicht sichtbaren Daten funktioniert.

Wir eliminieren alle Users, die weniger als 50 Bewertungen abgegeben haben --> damit wir genug Daten für den Train Test Split haben.

In [ ]:
```
users = df_movies_ratings["userId"].value_counts()[df_movies_ratings["userId"].value
```

Danach passen wir das Dataframe an nur mit den oben definierten Users.

In [ ]:
```
df_ratings_50 = df_movies_ratings[df_movies_ratings["userId"].isin(users)]
```

Für den Train Datensatz nehmen wir einen Sample von 80% der Daten. Die restlichen 20% werden dem Test Datensatz zugewiesen, indem alle Train Daten gelöscht werden.

In [ ]:
```
train = df_ratings_50.groupby("userId").sample(frac = 0.8, random_state = 42)
test = df_ratings_50.drop(train.index)
```

Wir erstellen hier eine User-Item-Matrix mit dem userId als Index und movieId als Spalten und die jeweiligen Filmbewertungen als Werte vom Trainings Datensatz.

Die Matrix wird dazu verwendet, die Beziehungen zwischen Users und Items darzustellen und zu analysieren, um personalisierte Empfehlungen für Benutzer zu generieren und Vorhersagen über ihre Bewertungen von Artikeln zu treffen.

In [ ]:
```
df_user_movie_matrix_train = train.pivot(index = "userId", columns = "movieId", valu
```

Jetzt wird aus dem User-Item-Matrix eine binäre User-Liked-Matrix erzeugt. Wir haben uns dazu für eine binäre User-Liked-Matrix entschieden, vor allem wegen der Einfachheit: Somit ist es für uns sehr einfach zu sehen, welche Filme einem User gefallen haben.

In der Explorativen Datenanalyse haben wir festgestellt, dass der Durchschnitt aller Bewertungen etwa bei einer 3 ist. Deswegen haben wir uns entschieden die 3 als Richtwert zu nehmen, sprich alle Filme mit den Bewertungen von über 3 gefielen dem User und alle unter oder gleich 3 gefielen dem User nicht.

Postitive Bewertungen (>3) werden zu 1 und negative Bewertungen (<=3) werden zu 0.

In [ ]:
```
df_user_movie_matrix_train[df_user_movie_matrix_train <= 3] = 0
df_user_movie_matrix_train[df_user_movie_matrix_train > 3] = 1
```

Out[ ]:

movieId	895	2679	4249	4484	5904	47237	47962	71677	72491	73319	...	209049	209051
userId													
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
19	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
75	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
84	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
162482	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
162515	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
162516	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
162521	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
162534	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0

13682 rows × 17541 columns

Beim Testdatensatz wird dasselbe gemacht wie oben beim Trainingsdatensatz.

In [ ]:
```
df_user_movie_matrix_test = test.pivot(index = "userId", columns = "movieId", values
```

In [ ]:
```
df_user_movie_matrix_test[df_user_movie_matrix_test <= 3] = 0
df_user_movie_matrix_test[df_user_movie_matrix_test > 3] = 1

df_user_movie_matrix_test
```

Out[ ]:

movieId	4484	47237	47962	71677	72491	73319	73321	73929	74131	74135	...	208765	20
userId													
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	
19	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	
75	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	
84	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	
162482	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	
162515	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	
162516	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	
162521	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	
162534	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	

13682 rows × 11978 columns

Wie viele Nullstellen gibt es in Train und Test?

In [ ]:
```
print(f"Percentage of 0's in train set: {round((df_user_movie_matrix_train == 0).sum
print(f"Percentage of 0's in test set: {round((df_user_movie_matrix_test == 0).sum()
```

```
Percentage of 0's in train set: 99.59%
Percentage of 0's in test set: 99.85%
```

Hier lesen wir nun alle gut bewerteten Filme pro User im Train-Set ein.

In [ ]:
```
# get all liked train movies per user
liked_movies_train = {}
for user in df_user_movie_matrix_train.index:
    liked_movies_train[user] = df_user_movie_matrix_train.columns[df_user_movie_matr
```

Das gleiche machen wir nun auch für das Test Datenset

In [ ]:
```
## get all liked test movies per user
liked_movies_test = {}
for user in df_user_movie_matrix_test.index:
    liked_movies_test[user] = df_user_movie_matrix_test.columns[df_user_movie_matrix
```

Um Vorschläge zu generieren, müssen wir zuerst Users entfernen, welche in einem der beiden Dictionaries, keine Filme besitzten. Für diese können wir nähmlich keine Filmvorschläge machen, da wir daraus kein Fazit ziehen können. Zunächst erstellen wir eine Liste welche diese User beinhalten.

In [ ]:
```
## get users which do not have any liked movies in test and train set
users_to_remove = []
for user in liked_movies_train:
    if len(liked_movies_train[user]) == 0 or len(liked_movies_test[user]) == 0:
        users_to_remove.append(user)
```

Nun entfernen wir die User, die keine Likes in Train und Test haben.

In [ ]:
```
for user in users_to_remove:
    liked_movies_train.pop(user, None)
    liked_movies_test.pop(user, None)
```

Da es sich bei den beiden Listen um MovieId's handelt, wir jedoch die Titel der Filme benötigen, tauschen wir diese um.

In [ ]:
```
# get name of movies in train dictionary
train_movies = {}
for user in liked_movies_train:
    train_movies[user] = df_movies[df_movies["movieId"].isin(liked_movies_train[user

# get name of movies in test dictionary
test_movies = {}
for user in liked_movies_test:
    test_movies[user] = df_movies[df_movies["movieId"].isin(liked_movies_test[user])
```

Um nun das ganze ein wenig schneller zu fertigen, nehmen wir pro User im Trainset, bloss die ersten 10 Filme

In [ ]:
```
for user in train_movies:
    train_movies[user] = list(set(train_movies[user]))[:10]
```

Testing auf Userprofile

Um die Effektivität unserer Recommender Systems zu messen verwenden wir die Metrik "Hits"

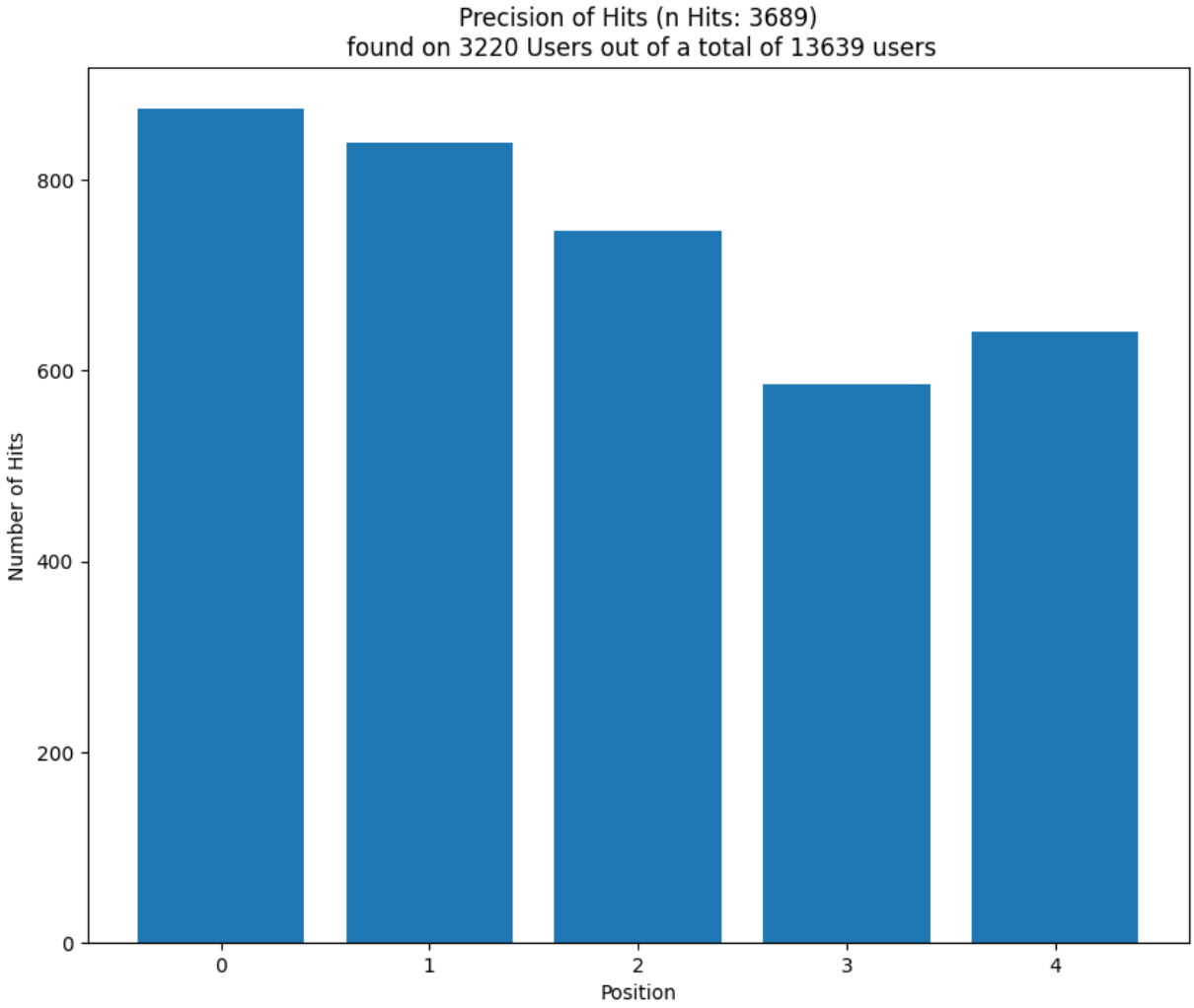
Im Kontext eines Recommender Systems beziehen sich "Hits" auf die Filme, die das System einem User empfiehlt. Dabei zählen wir wie viele Empfehlungen tatsächlich vom User "getroffen" werden.

Nachdem wir die Daten vorbereitet haben, müssen wir nun den Test pro Userprofile erstellen. Dazu werden pro Recommender allen Users, basierend auf den 10 gemochten Filmen, Vorschläge generiert. Diese Vorschläge werden dann mit dem Test-Set verglichen und es werden die Hits gezählt.

TF-IDF

In [ ]:
```
# runs 12 mins
testing_tfidf = tfidf_rec_user(df_movies_nlp, train_movies, cols)
```

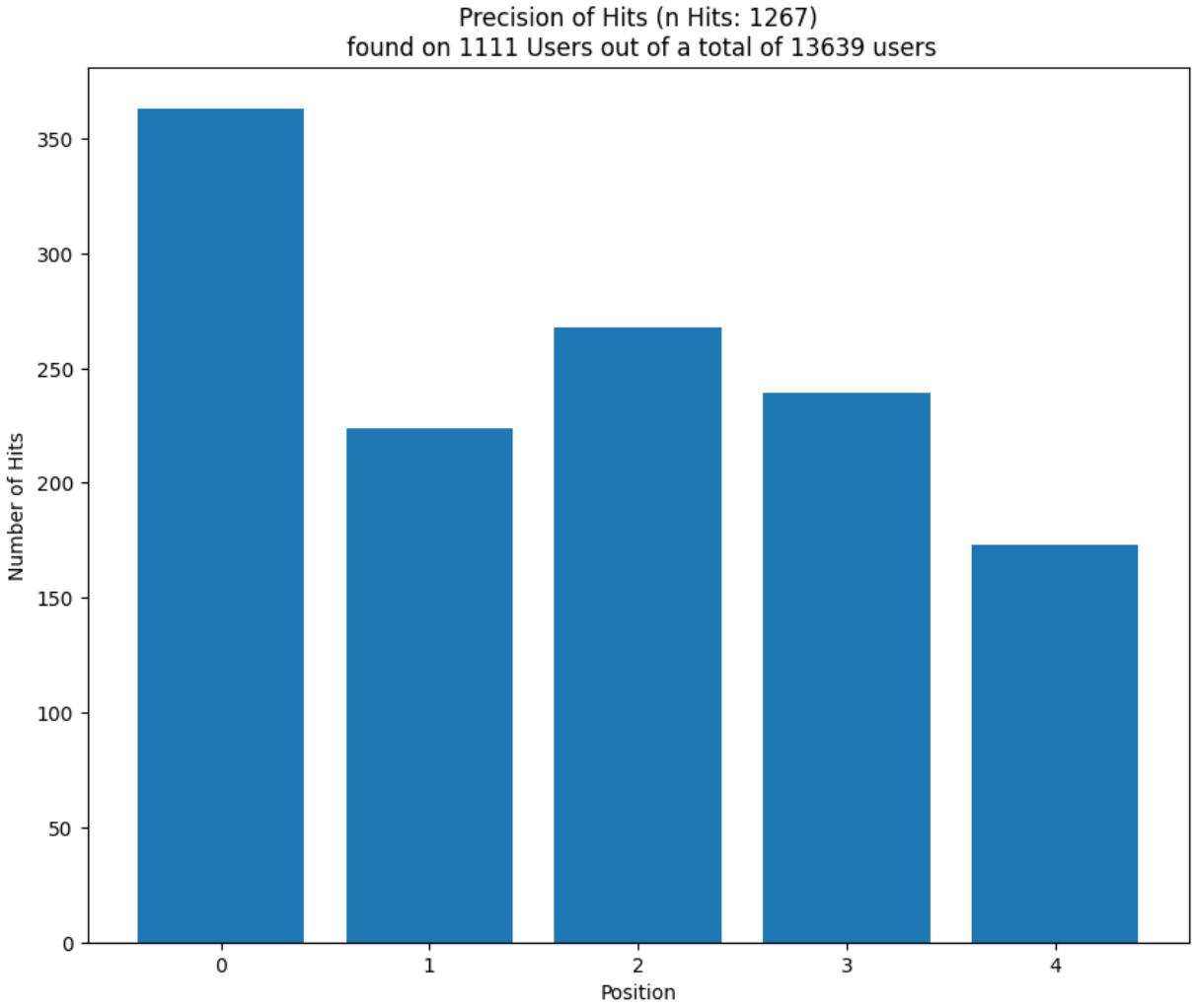
In [ ]:
```
calculate_hits(testing_tfidf, test_movies)
```



LSA

In [ ]:
```
# runs about 50 mins
testing_lsa = lsa_rec_user(df_movies_nlp, train_movies, cols)
```

In [ ]:
```
calculate_hits(testing_lsa, test_movies)
```



Doc2Vec

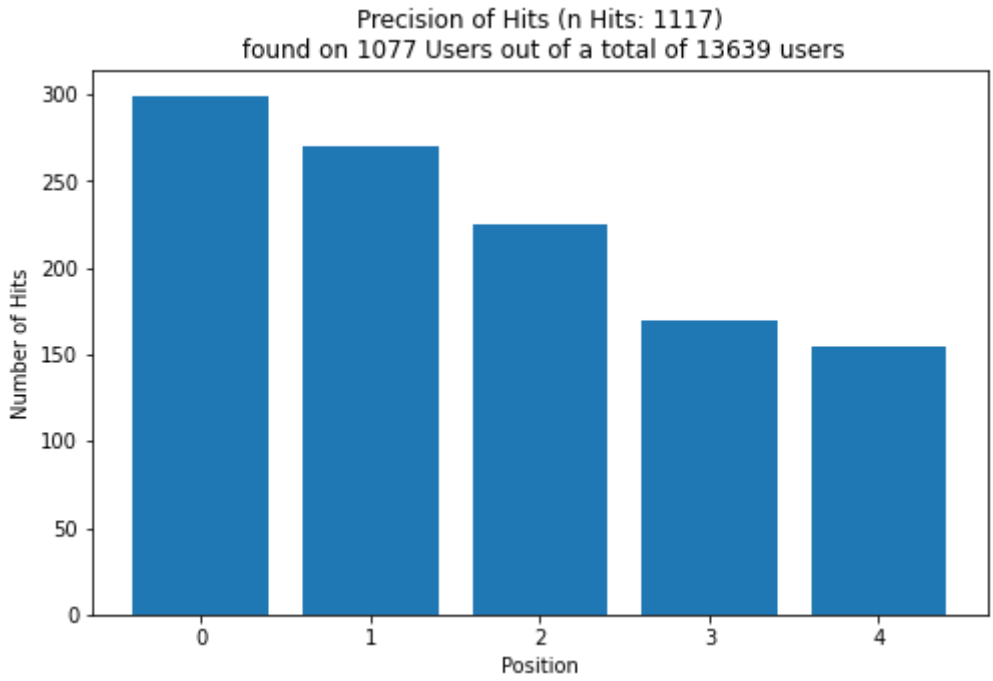
In [ ]:
```
# runs about 2.5 hours
testing_doc2vec = doc2vec_rec_user(df_movies_nlp, train_movies, cols)
```

In [ ]:
```
# export testing_doc2vec to csv
pd.DataFrame.from_dict(testing_doc2vec, orient='index').to_csv("testing_doc2vec.csv"
```

In [ ]:
```
# import testing_doc2vec from csv
dict_testing_doc2vec = pd.read_csv("testing_doc2vec.csv", index_col=0).to_dict(orien
```

In [ ]:
```
d2v_dict = dict_testing_doc2vec
d2v_dict = {k: [v for v in inner_dict.values()] for k, inner_dict in dict_testing_do
```

In [ ]:
```
calculate_hits(d2v_dict, test_movies)
```



Fazit

- TF-IDF: 3689 Hits auf 3220 Users
- LSA: 1267 Hits auf 1111 Users
- Doc2Vec: 1117 Hits auf 1077 Users

Basierend auf den Evaluierungsergebnissen scheint es, dass der TF-IDF-Algorithmus die höchste Anzahl an Hits bei den meisten Users hatte. Dies bedeutet, dass es am effektivsten war, Usern ähnliche Filme zu empfehlen. Der LSA-Algorithmus hatte die zweithöchste Anzahl an Hits, jedoch bei einer geringeren Anzahl von User. Der Doc2Vec-Algorithmus hatte die wenigsten Hits, aber bei einer ähnlichen Anzahl von User wie der LSA-Algorithmus. Insgesamt scheint der TF-IDF-Algorithmus der beste Performer unter den drei getesteten Algorithmen zu sein.