IS675 - Deep Learning for Business

Fall 2024

Instructor Name: Dr. Mostafa Amini

GRASP - Graph-Based Hybrid Recommender Analyzing Sentiment Patterns

Team - 9

Robert Torres - 025525719

Kishan Mehta - 032241584

Vedant Telrandhe - 032179119

Akshitha Chittipolu - 032180952

Table of Contents

1. INTRODUCTION

Rent the Runway (RTR) is a trailblazing fashion rental service that allows users to rent, subscribe, or buy designer clothing and accessories. Launched in 2009, the company disrupted traditional fashion consumption by introducing a shared economy model, making luxury and high-end fashion more accessible and sustainable. RTR appeals to fashion-conscious individuals who wish to enjoy variety without committing to permanent purchases, reducing wardrobe clutter and promoting eco-friendly consumption.[1]

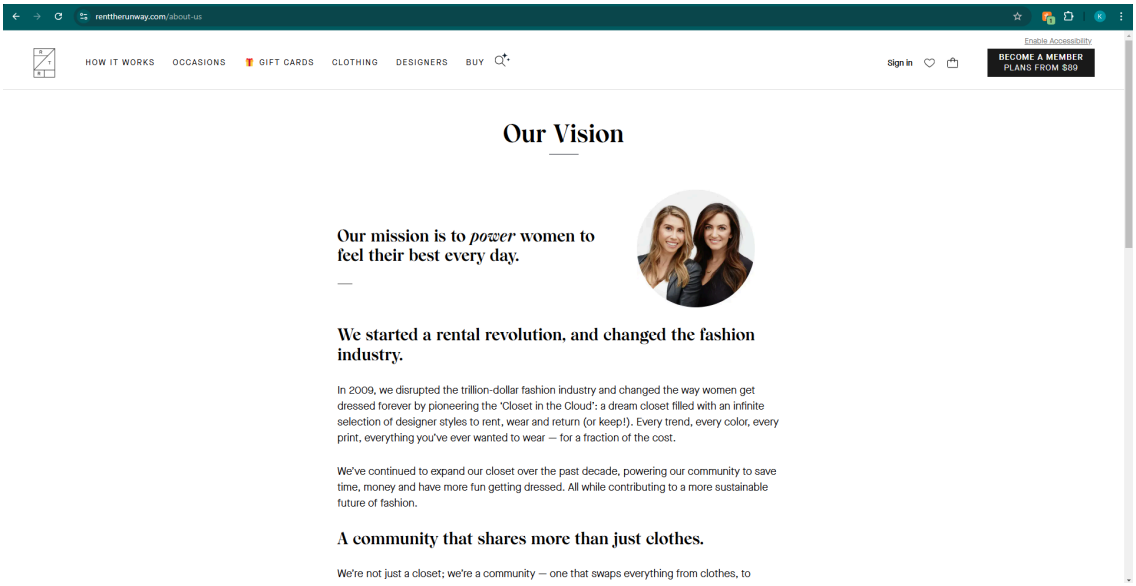

Fig 1. Renttherunway Website

Users browse the RTR website or mobile app, select items, and indicate the rental period. Once rented, items are delivered to the customer's door. After use, customers return the items using pre-paid packaging. RTR takes care of cleaning and maintaining the items. Rent the Runway operates through three primary services:

1. **One-Time Rentals:** Customers can rent individual pieces for specific events or needs, choosing from a wide range of designer dresses, gowns, and accessories.

2. **Subscription Plans:** RTR offers flexible subscription plans that allow members to rent a set number of items per month. Subscribers can swap items frequently, enabling a dynamic and refreshed wardrobe.
3. **Purchasing Options:** Customers also have the option to buy pre-loved or new items at discounted prices.

Customers can browse an extensive inventory of over 800 designers, order online or via the RTR app, and receive doorstep delivery with pre-paid packaging for hassle-free returns. The company manages professional cleaning and maintenance of all rented items, ensuring consistent quality and convenience.

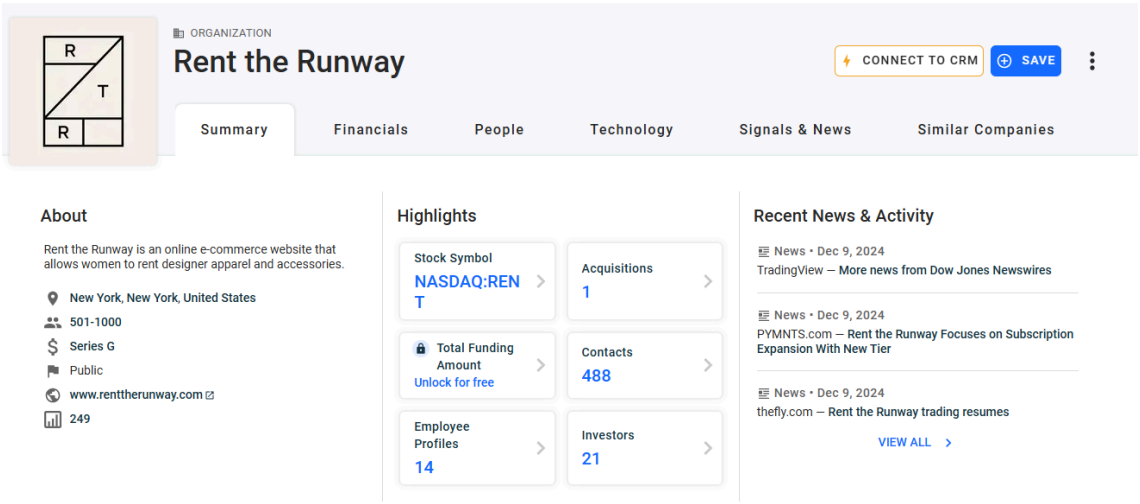

Fig 2. Renttherunway's Crunchbase profile

RTR's focus on sustainability is evident in its efforts to reduce overproduction and promote eco-friendly consumption. It also addresses common customer concerns through features like fit guarantees, which include a complimentary second size, and detailed user reviews with photos for informed decision-making. Customer feedback plays a pivotal role in RTR's operations, with reviews and ratings providing insights for inventory curation, quality improvements, and personalized recommendations powered by AI. Leveraging advanced deep learning technologies, RTR analyzes customer sentiment using natural language processing, predicts demand to

optimize inventory, and enhances search and recommendation systems through image recognition. With its innovative blend of fashion, technology, and customer-centric design, Rent the Runway offers a compelling case study for deep learning applications, particularly in personalized experiences, trend forecasting, and operational efficiency.

2. PROBLEM STATEMENT

In 2023, a young woman named Claire shared her unfortunate experience with ill-fitting clothes during one of the most important days of her life—her first job interview after graduating from college. Claire had spent weeks preparing for the interview at a prestigious law firm, and she wanted to make the best possible impression. She purchased a professional suit online, carefully selecting what she believed was her correct size based on the retailer's sizing chart. When the suit arrived, it seemed slightly tight, but Claire brushed off her discomfort, thinking it would stretch out after wearing it for a while. On the day of her interview, she put on the suit and immediately felt constricted. The jacket was too tight across her shoulders, and the pants pinched at her waist. Still, she was determined to look polished and professional. As soon as Claire arrived at the interview, things started to go wrong. Sitting down in the too-small suit was uncomfortable, and halfway through answering a question from one of the partners, she felt a sharp tear—the seam of her pants had split open. Horrified, Claire tried to keep her composure, but she could feel her confidence slipping away. The rest of the interview was a blur as she anxiously tried to hide the tear and avoid moving too much. Afterward, Claire left the office feeling humiliated. She didn't get the job, and while she couldn't be sure if it was because of her performance or the wardrobe malfunction, it left her feeling defeated. Reflecting on the experience later, Claire realized how much pressure she had placed on herself to fit into clothes that weren't right for her body. The stress of wearing an ill-fitting suit had not only affected her comfort but also her confidence during one of the most important moments of her career. This story serves as a reminder that wearing clothes that fit properly is not just about appearance—it can have real consequences on how we feel and perform in critical situations. This anecdote illustrates how something as simple as wearing clothes that don't fit can lead to significant emotional distress and even impact professional opportunities.

The challenge of accurately fitting clothes is a significant issue in the fashion industry, particularly for e-commerce platforms where customers rely on online size charts and reviews to make purchasing decisions.[2] Claire's story illustrates the emotional and professional consequences of wearing ill-fitting clothing, which can lead to discomfort, embarrassment, and reduced confidence. For businesses like Rent the Runway (RTR), this issue not only impacts

customer satisfaction but also increases return rates and operational costs. While RTR collects extensive data on customer feedback, including fit-related reviews and measurements, there is a need for robust analytical methods to derive actionable insights from this data and provide reliable size recommendations.

Deep learning methods can play a pivotal role in addressing the problem of inaccurate fit predictions by leveraging the wealth of data RTR has collected. Sentiment analysis using Natural Language Processing (NLP) techniques, such as Convolutional Neural Networks (CNNs) and pre-trained encoders such as BERT, can analyze unstructured customer reviews to classify fit-related sentiments (e.g., true-to-size, larger, or smaller). These models can extract nuanced insights from textual feedback, helping businesses understand patterns and biases in clothing fit. Furthermore, keyword extraction can align customer reviews with product-specific attributes, enabling personalized recommendations based on historical feedback.

By implementing deep learning-based methods, RTR can significantly improve customer satisfaction and confidence in their size recommendations. Sentiment-driven insights allow customers to make informed decisions quickly, reducing the frustration of trial-and-error shopping. Tailored recommendations not only enhance the shopping experience but also minimize return rates, leading to operational cost savings and increased trust in the platform. Ultimately, integrating deep learning techniques into RTR's fit prediction system aligns customer expectations with delivered outcomes, creating a seamless and satisfying user experience while positioning the company as a leader in personalized fashion services. [3]

2.1. Structured Data :

Structured data, consisting of well-defined rows and columns, is highly useful in deep learning as it provides organized, quantitative information that can be directly fed into machine learning and deep learning models for analysis and predictions. Structured data in this project refers to the well-organized, tabular information present in the dataset, such as customer demographics, product characteristics, and transaction details. These data points are numeric or categorical and are easily stored and analyzed in a relational database.

The dataset contains structured data in JSON format with 192,544 rows and a total size of 123 MB. The features include customer-specific details **(e.g., user_id, age, body type, bust size, height, and weight), product information (e.g., item_id, category, and size), and feedback data (e.g., rating, fit, review_text, review_summary, and review_date**). This structured data is complemented by unstructured text in the form of reviews, providing qualitative feedback on customer experiences with rented clothing. This structured format allows for straightforward analysis, such as examining the correlation between variables like height and fit or exploring customer preferences across different categories. Using this data, the project aims to develop predictive models and create visualizations that reveal patterns, ultimately enhancing the recommender system's ability to suggest optimal sizes for customers.

Deep learning models like Neural Networks can take structured data as input. For instance, customer attributes (**height, weight, age**) and product features (**category, size**) can be encoded numerically and fed into these models. The layers of the network extract and learn hierarchical patterns in the data, enabling accurate predictions. Deep learning models trained on structured data can handle diverse applications such as anomaly detection, customer segmentation, and prediction tasks at scale. Once trained, these models can automate processes like personalized recommendations or operational forecasts.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
fit	user_id	bust_size	item_id	weight	rating	rented_fo	review_te	body_typ	review_su	category	height	size	age	review_da	bra_size	cup_size	weightnar	ratingnan	heightnan	agenan
fit	420272	34d	2260466	137	10	vacation	An adorab	hourglass	So many c	romper	172.72	14	28	April 20, 2(	34	d	0	0	0	0
fit	273551	34b	153475	132	10	other	I rented th	straight &	I felt so gl	gown	167.64000	12	36	June 18, 2(	34	b	0	0	0	0
fit	360448	Missing	1063761	135	10	party	This hugge	Missing	It was a gr	sheath	162.56	4	116	December	34	Missing	1	0	0	0
fit	909926	34c	126335	135	8	formal aff	I rented th	pear	Dress arriv	dress	165.1	8	34	February 1	34	c	0	0	0	0
fit	151944	34b	616682	145	10	wedding	I have	athletic	Was in lov	gown	175.26	12	27	Septembe	34	b	0	0	0	0
fit	734848	32b	364092	138	8	date	Didn't actu	athletic	Traditiona	dress	172.72	8	45	April 30, 2(	32	b	0	0	0	0
fit	336066	34c	568429	112	10	everyday	This dress	hourglass	LITERALLY	dress	160.02	4	27	December	34	c	0	0	0	0
fit	86661	34d+	130259	118	10	formal aff	Fit was gre	full bust	Great dres	dress	160.02	8	65	January 1,	34	d	0	0	0	0
fit	166228	36d	1729232	135	10	formal aff	I was nerv	full bust	Great for t	gown	167.64000	21	27	June 27, 2(	36	d	1	0	0	0
fit	154309	32b	1729232	114	10	formal aff	The dress	petite	This dress	gown	160.02	1	33	October 1	32	b	0	0	0	0
small	185966	34b	1077123	135	8	party	The dress	athletic	It was fun	dress	160.02	12	33	January 2,	34	b	0	0	0	0
large	533900	34b	130259	135	8	wedding	This dress	pear	Stunning d	dress	167.64000	8	30	January 7,	34	b	0	0	0	0
fit	87660	36a	1295171	120	10	party	The S was	straight &	This dress	sheath	167.64000	8	26	July 28, 20	36	a	0	0	0	0
large	391778	36d	143094	142	8	party	I ordered t	apple	Ordered th	dress	157.48000	20	29	October 4,	36	d	0	0	0	0
fit	721308	34b	123793	118	10	formal aff	Fit great, s	athletic	Stunning g	gown	165.1	2	32	May 29, 2(	34	b	0	0	0	0
fit	829124	34c	2595752	140	10	party	Medium w	hourglass	These legg	leggings	170.18	20	30	May 16, 2(	34	c	0	0	0	0
fit	499943	34c	909221	170	10	vacation	This dress	pear	Really cute	dress	172.72	20	35	March 28,	34	c	0	0	0	0
fit	339899	34d	1622747	143	10	party	Little tight	athletic	LOVED	dress	165.1	12	26	Novembe	34	d	0	0	0	0
fit	649288	34b	172027	115	8	wedding	This is a pr	petite	Semi-form	dress	160.02	8	28	July 3, 201	34	b	0	0	0	0
small	16800	34c	1229740	135	10	formal aff	super cute	petite	gala ready	gown	167.64000	21	33	Novembe	34	c	0	0	0	0
fit	661150	36d	900878	145	10	wedding	I rented th	hourglass	Perfect for	dress	165.1	12	33	May 23, 2(	36	d	0	0	0	0
fit	983550	32d+	197391	140	8	other	I wore this	hourglass	Fashionab	dress	167.64000	12	30	February 2	32	d	0	0	0	0
fit	871659	36c	2736018	135	10	work	Super	pear	Wore to w	top	165.1	20	32	March 16,	36	c	1	0	0	0
fit	497541	34c	382883	120	8	work	I loved eve	apple	Well made	dress	167.64000	1	21	February 1	34	c	0	0	0	0
fit	862005	32c	134393	140	10	wedding	It fit perfe	pear	Great dres	dress	172.72	8	27	June 3, 20	32	c	0	0	0	0

Fig 3. Dataset

2.2. Unstructured Data :

Unstructured data, such as text, images, audio, and videos, lacks a predefined format but constitutes a significant portion of the data generated globally. Deep learning, with its ability to process and extract meaningful patterns from raw, unstructured data, has revolutionized its utilization across industries. For instance, in Natural Language Processing (NLP), unstructured textual data like customer reviews or social media posts can be analyzed to uncover sentiments, trends, and user preferences. Similarly, in computer vision, unstructured image data can be used for tasks like object detection, facial recognition, and image classification. Deep learning excels in this domain because of its capacity to automatically learn and represent complex features through neural networks without the need for extensive manual feature engineering.

Unstructured data in this project consists of the textual reviews provided by customers in fields like **review_text** and **review_summary**. Unlike structured data, these reviews do not follow a predefined format, making them harder to analyze directly. However, they are rich in qualitative insights, offering detailed feedback about fit, comfort, and overall satisfaction. Sentiment analysis and Natural Language Processing (NLP) techniques are applied to this data to extract valuable information about product fit (e.g., whether an item is "**too small**" or "**perfect"**). By analyzing this unstructured data, the project provides a deeper understanding of customer sentiments, complements the structured data analysis, and enhances the personalization of size recommendations. Recommender systems, such as collaborative filtering and content-based filtering, traditionally rely on structured data like user-item interactions (**ratings, purchases**). However, integrating unstructured data, such as textual reviews or product descriptions, significantly enhances their accuracy and personalization.

Textual features from reviews or product descriptions are extracted using NLP models. Advanced methods like Transformer-based embeddings (e.g., BERT, GPT) or simple vectorization techniques (e.g., TF-IDF) transform textual content into numerical feature spaces. These embeddings provide a detailed understanding of product attributes, enabling the system to match items with similar descriptions or customer feedback.[3] A user's textual reviews are analyzed to infer preferences, creating a personalized profile. This profile is then matched with

items that have similar attributes based on textual content. For instance, if a user frequently praises "durable material" or "elegant design," products with these characteristics in their descriptions or reviews will be recommended.

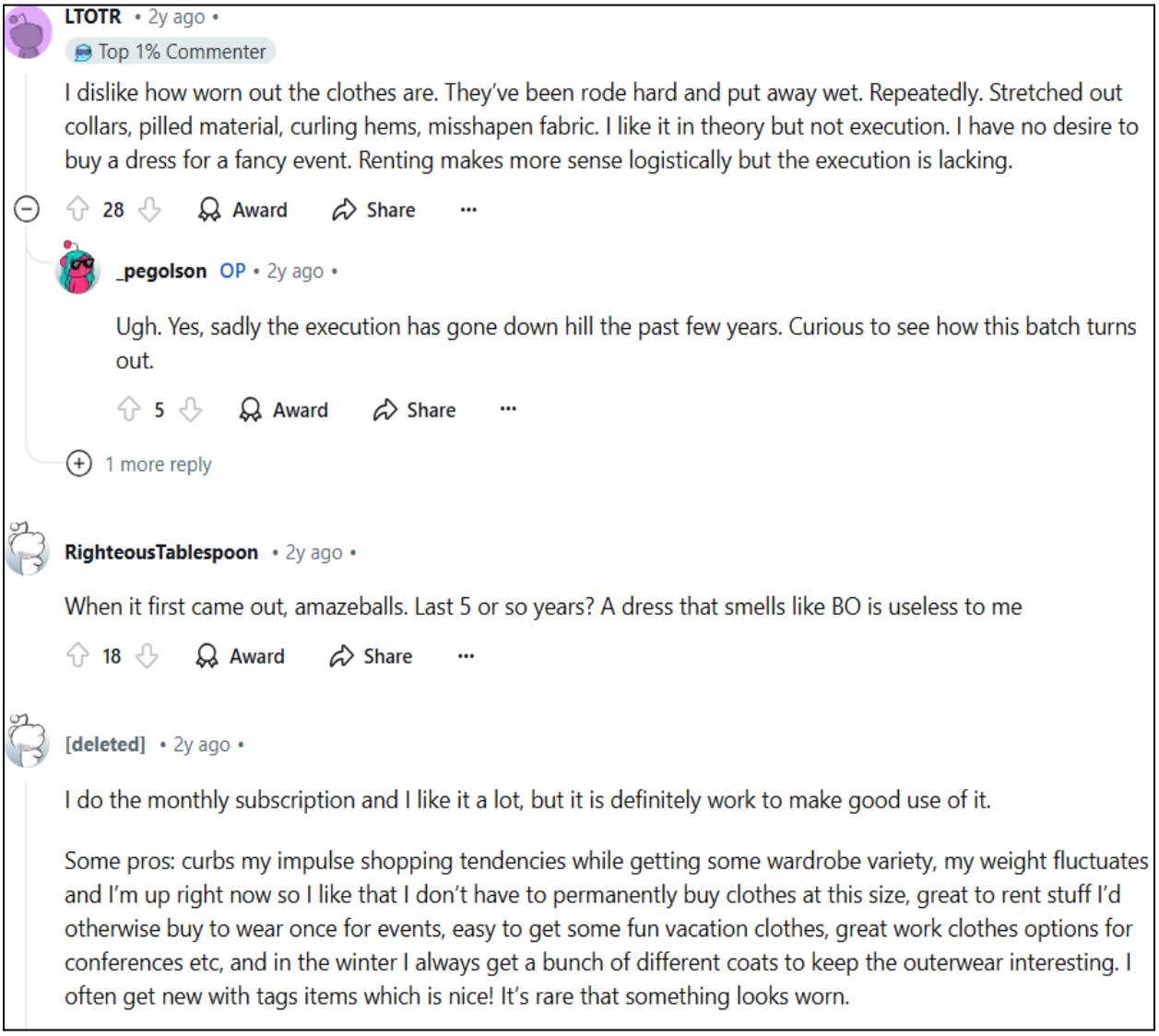

Fig 4. RTR's online reviews

3. Data Understanding and Visualization

We explored the dataset to understand its structure, quality, and key characteristics. This allowed us to identify potential issues like missing values, outliers, or imbalances and to better understand the relationships between features.

3.1. Exploring the data:

```
dataset = pd.read_json(r'C:\Users\kisha\Downloads\renttherunway_final_data.json', lines=True)
print(dataset.shape)
```

```
(192544, 15)
```

```
dataset.head(20)
```

	fit	user_id	bust size	item_id	weight	rating	rented for	review_text	body type	review_summary	category	height	size	age	review_date
0	fit	420272	34d	2260466	137lbs	10.0	vacation	An adorable romper! Belt and zipper were a lit...	hourglass	So many compliments!	romper	5' 8"	14	28.0	April 20, 2016
1	fit	273551	34b	153475	132lbs	10.0	other	I rented this dress for a photo shoot. The the...	straight & narrow	I felt so glamourous!!!	gown	5' 6"	12	36.0	June 18, 2013
2	fit	360448	NaN	1063761	NaN	10.0	party	This hugged in all the right places! It was a ...	NaN	It was a great time to celebrate the (almost) ...	sheath	5' 4"	4	116.0	December 14, 2015
3	fit	909926	34c	126335	135lbs	8.0	formal affair	I rented this for my company's black tie award...	pear	Dress arrived on time and in perfect condition.	dress	5' 5"	8	34.0	February 12, 2014
4	fit	151944	34b	616682	145lbs	10.0	wedding	I have always been petite in my upper body and...	athletic	Was in love with this dress !!!	gown	5' 9"	12	27.0	September 26, 2016
5	fit	734848	32b	364092	138lbs	8.0	date	Didn't actually wear it. It fit perfectly. The...	athletic	Traditional with a touch a sass	dress	5' 8"	8	45.0	April 30, 2016
6	fit	336066	34c	568429	112lbs	10.0	everyday	This dress is so sweet. I loved the print. The...	hourglass	LITERALLY THE CUTEST DRESS EVER	dress	5' 3"	4	27.0	December 7, 2017
7	fit	86661	34d+	130259	118lbs	10.0	formal affair	Fit was great. Maybe a little tight under the ...	full bust	Great dress, beautifully made. I received lot...	dress	5' 3"	8	65.0	January 1, 2013
8	fit	166228	36d	1729232	NaN	10.0	formal affair	I was nervous of it looking cheap when it arri...	full bust	Great for black tie event!	gown	5' 6"	21	27.0	June 27, 2016
9	fit	154309	32b	1729232	114lbs	10.0	formal affair	The dress was very flattering and fit perfect!...	petite	This dress was everything! It was perfect for ...	gown	5' 3"	1	33.0	October 17, 2016
10	small	185966	34b	1077123	135lbs	8.0	party	The dress arrived with a small hole in the bea...	athletic	It was fun to wear a dress I wouldn't normally...	dress	5' 3"	12	33.0	January 2, 2018

Fig 5. Understanding the data

The dataset comprises **192,544 rows** and **15 features**, offering a detailed view of customer interactions with Rent the Runway's platform. Key columns include **user details** (e.g., age, height, weight, and bust size), product attributes (e.g., category and size), and user feedback, such as **ratings**, **review summaries**, and **body type**.

The **"fit"** column provides customer feedback (e.g., "fit" or "small"), which can directly inform recommendations. The **"rented for"** column categorizes use cases like "vacation," "formal affair," or "wedding," giving us insights into customer intent.

```
: dataset.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 192544 entries, 0 to 192543
Data columns (total 15 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   fit             192544 non-null  object
 1   user_id         192544 non-null  int64
 2   bust_size       174133 non-null  object
 3   item_id         192544 non-null  int64
 4   weight          162562 non-null  object
 5   rating          192462 non-null  float64
 6   rented_for      192534 non-null  object
 7   review_text     192544 non-null  object
 8   body_type       177907 non-null  object
 9   review_summary  192544 non-null  object
 10  category        192544 non-null  object
 11  height          191867 non-null  object
 12  size            192544 non-null  int64
 13  age             191584 non-null  float64
 14  review_date     192544 non-null  object
dtypes: float64(2), int64(3), object(10)
memory usage: 22.0+ MB
```

Fig 6: Null Value Count

The above table states missing values across several key features in the dataset. For instance, **bust_size** has a significant number of missing entries, which might indicate that users often skip providing this detail, maybe due to its sensitive nature. Similarly, weight has an even higher proportion of missing values, suggesting a similar trend where users might feel hesitant to share personal information. On the other hand, features like **rating** and **body_type** show fewer missing entries, stating that users are more likely to provide feedback on their experience and general attributes. The height column has minor gaps, which could result from incomplete customer profiles during data collection. The textual columns, such as **review_text** and **review_summary**, are complete, giving rich insights for sentiment analysis. The data types show a mix of numerical and categorical variables.

3.2 Finding percentage of all the Missing Values:

```
: ## Here we will check the percentage of nan values present in each feature
  ## 1 -step make the list of features which has missing values
  features_with_na=[features for features in dataset.columns if dataset[features].isnull().sum()>1]

  ## 2- step print the feature name and the percentage of missing values
  for feature in features_with_na:
      print(feature, '->', (dataset[feature].isnull().sum()/192544*100),  '% Missing Values')
```

```
bust_size -> 9.561970250955627 % Missing Values
weight -> 15.571505733754362 % Missing Values
rating -> 0.04258766827322586 % Missing Values
rented_for -> 0.005193618082100715 % Missing Values
body_type -> 7.601898786770817 % Missing Values
height -> 0.3516079441582184 % Missing Values
age -> 0.49858733588166865 % Missing Values
```

Fig 7: Missing Values Percentage

Now we have our missing value percentage, The feature bust_size has 9.56% missing values, which indicates that a significant portion of users have not provided this detail. This missing data may affect personalized recommendations as bust_size is closely tied to fit-related insights. Similarly, the weight feature shows the highest percentage of missing values at 15.57%.

The feature body_type has 7.6% missing values, which might suggest users either skipped this field or found it hard to categorize themselves. Features like rating and rented_for have negligible missing values (0.04% and 0.005%, respectively) and can be easily filled without much impact on analysis.

3.3. Categorical Features:

```
The feature is fit and number of categories are 3
The feature is bust_size and number of categories are 107
The feature is weight and number of categories are 191
The feature is rented_for and number of categories are 10
The feature is body_type and number of categories are 8
The feature is category and number of categories are 68
The feature is height and number of categories are 25
```

Fig 8: Features and number of categories

The dataset has several categorical features, each with a different number of unique values. The fit feature is simple with just 3 categories, showing how well the item fit. The

bust_size feature has 107 unique values, and weight has 191, showing a lot of variety in customer inputs. The rented_for feature includes 10 categories, representing different occasions like weddings or parties. The body_type feature has 8 categories, describing different body shapes. The category feature, which lists the types of items, has 68 unique values, showing a wide range of products. Finally, the height feature has 25 unique values, reflecting the different height ranges of customers. These features give us detailed information about customer preferences and product variety, which is very useful for creating a recommendation system that works well for different types of people.

3.4. Distribution of Categorical Features:

The plots help us understand how different features are distributed based on the fit categories: fit, small, and large. This breakdown gives us valuable insights into customer feedback and patterns.

a. Fit Distribution: Most customers found their rented items to fit well, with the fit category clearly dominating. However, there are still noticeable reports of items being too small or large, suggesting that there is room for improvement in sizing accuracy.

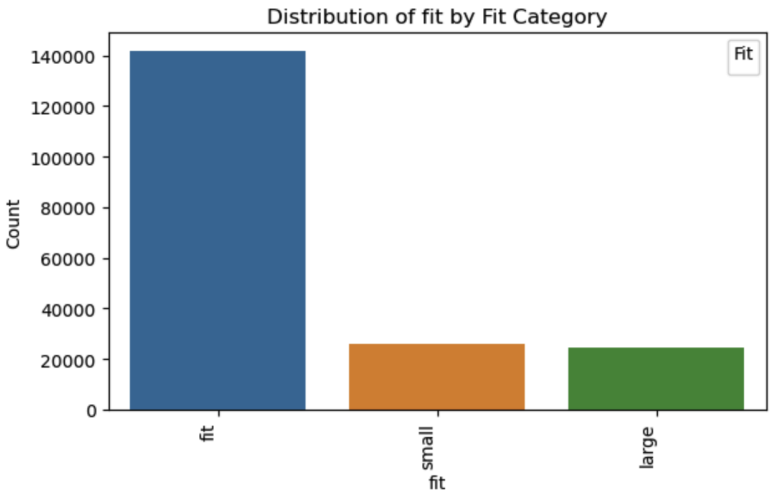

Fig 9: Distribution of Fit by Fit Category

b. Bust Size: The bust_size feature shows a wide variety of values. For most sizes, items tend to fit well, but there are consistent reports of items being too small or large across all bust sizes. This suggests that fit issues occur across a broad range of body measurements, indicating the need for a more tailored approach to sizing.

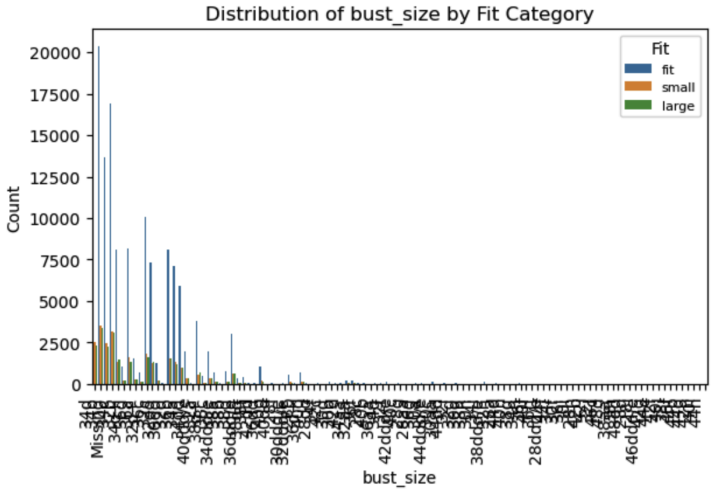

Fig 10: Distribution of bust_size by Fit Category

c. Weight: The distribution of weight by fit category shows that most customers, regardless of their weight, experienced items that fit well. However, extreme weight ranges show more reports of small or large fits. This could mean that edge cases in body weight are not being fully accounted for by current sizing standards.

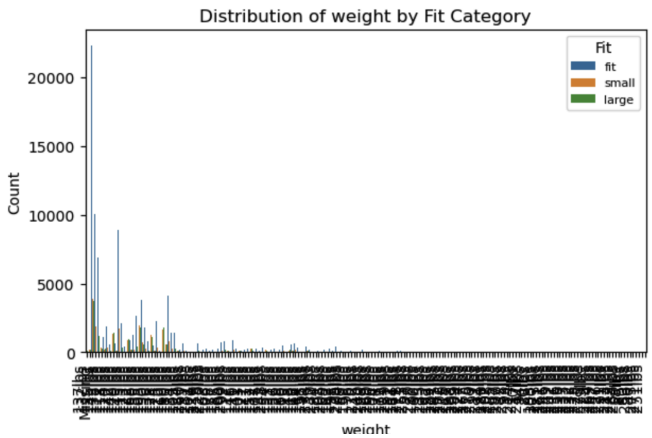

Fig 11: Distribution of weight by Fit Category

d. Rented For: When we look at why customers rented their items, most rentals for weddings, formal affairs, and parties fit well. However, there are noticeable fit issues for special occasions, which could be due to higher expectations for these events.

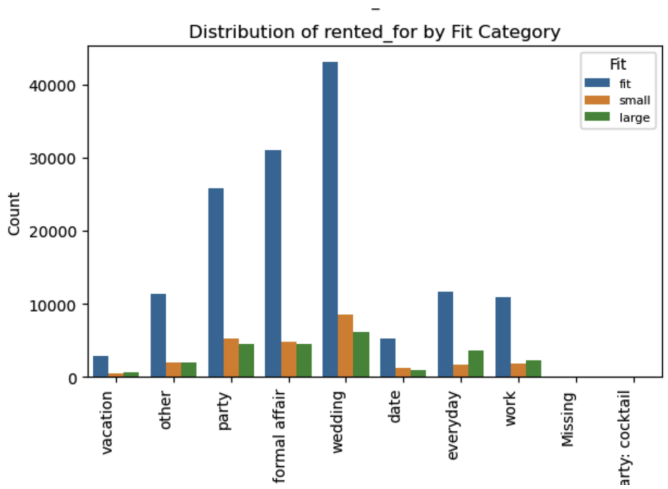

Fig 12: Distribution of rented_for by Fit Category

e. Body Type: Customers with body types like hourglass, athletic, and pear reported the best fit results, with the majority of items falling into the fit category. However, customers with less common body types, like apple or full bust, reported more fit issues. This suggests that some body shapes may not be as well-supported by current sizing options.

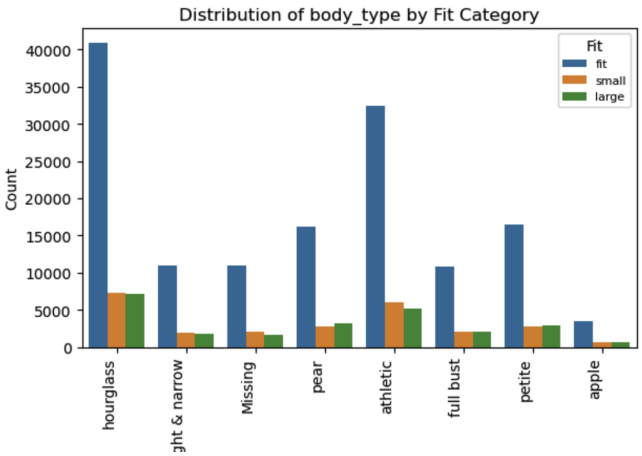

Fig 13: Distribution of body_type by Fit Category

f. Product Category: The category feature shows a broad range of items, with most fitting well. However, categories like dresses and gowns have higher numbers of fit issues, likely due to their more tailored designs.

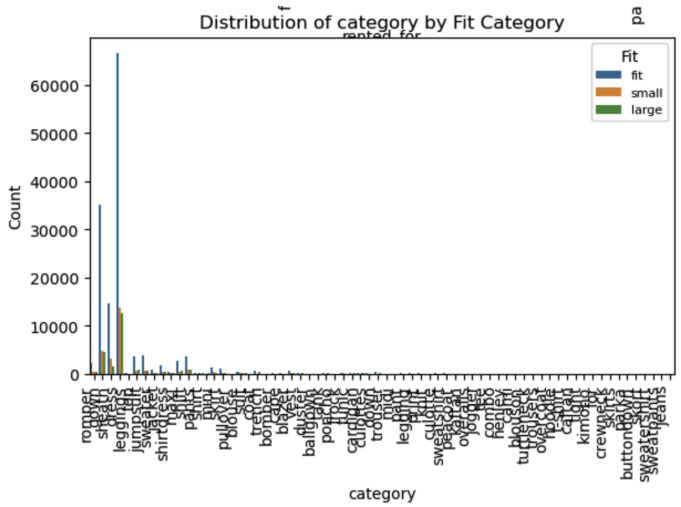

Fig 14: Distribution of category by Fit Category

g. Height: Most customers in common height ranges, such as 5’4” to 5’8”, reported items fitting well. However, customers who are much shorter or taller than average reported more issues with small or large fits.

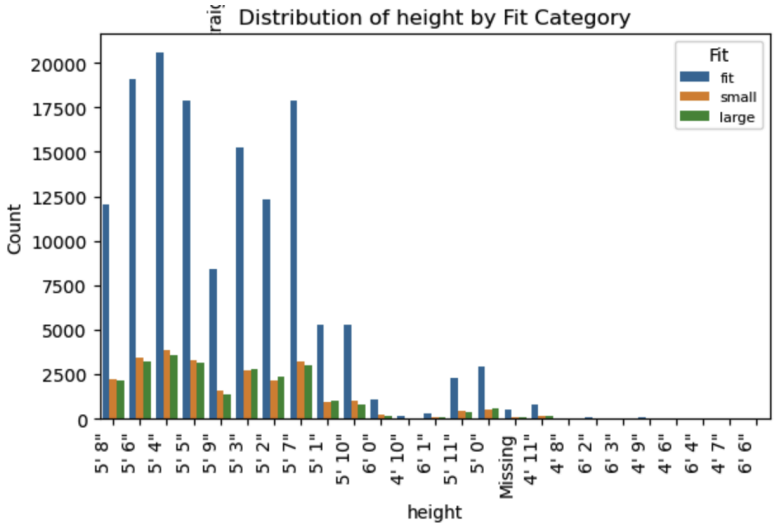

Fig 15: Distribution of height by Fit Category

3.5 The Outliers in Numerical Features:

Outliers are extreme values in numerical features that can affect data analysis and model performance. In this project, we decided to keep all outliers instead of removing or adjusting them. This decision was made to ensure the data reflects real-world scenarios, including rare or unusual cases.

Visual Analysis from the Plots:

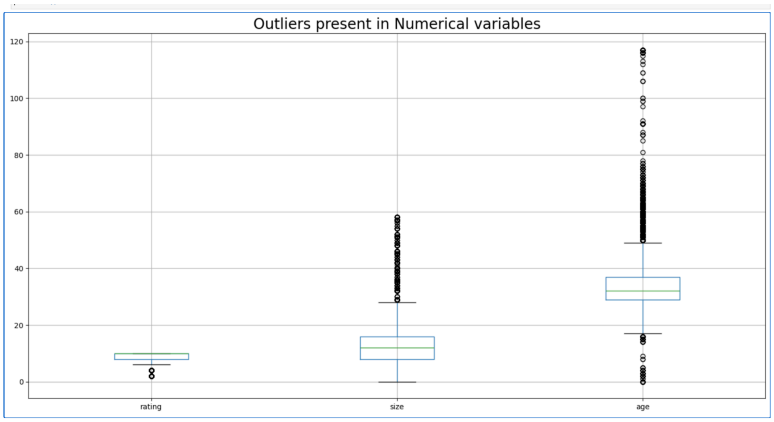

Fig 16: Outliers in Numerical Variables

The boxplots show noticeable outliers in features like size and age. For example, the age feature has values above 100, which could be errors or represent uncommon situations. These outliers extend far beyond in the boxplots, indicating a wide range of variability in the data. By keeping these outliers, we allow the recommendation system to consider all types of customer data, including edge cases.

3.6. Examination of correlation pairs:

We created a correlation matrix and printed the top 10 highest correlation pairs to better understand the relationships between features. Instead of using heatmaps, which would become cluttered due to the large number of features, we focused on analyzing these top correlations for clearer insights.

```
fit_fit                   fit_small                   0.659520
fit_small                 fit_fit                     0.659520
fit_fit                   fit_large                   0.643596
fit_large                 fit_fit                     0.643596
category_dress            category_gown               0.528371
category_gown             category_dress              0.528371
                          rented_for_formal affair    0.499655
rented_for_formal affair  category_gown               0.499655
height                    body_type_petite            0.398845
body_type_petite          height                      0.398845
body_type_athletic        body_type_hourglass         0.343992
body_type_hourglass       body_type_athletic          0.343992
rented_for_wedding        rented_for_formal affair    0.337475
rented_for_formal affair  rented_for_wedding          0.337475
category_dress            category_sheath             0.322374
dtype: float64
```

Fig 17: Top 10 Correlation Pairs

From the analysis, we observed that the fit categories (e.g., fit_fit, fit_small, fit_large) are moderately correlated with each other, with correlations around 0.65. This shows a pattern in how customers perceive fit, where these categories are somewhat connected. We also found that product categories, such as category_dress and category_gown, have a correlation of 0.528. Similarly, rented_for_formal affair and category_gown have a correlation of 0.499, reflecting that gowns are commonly chosen for formal events. We observed a correlation of 0.398 between body_type_petite and height, which makes sense as petite body types are often associated with shorter heights. Another pair is rented_for_formal affair and rented_for_wedding, with a correlation of 0.337, showing a connection between events requiring similar types of attire. This correlation analysis provides valuable insights into feature relationships, helping us understand how customer attributes, product types, and occasions interact.

4. Feature Engineering

4.1. Extracting features from a variable:

We worked on breaking down the bust_size feature into two separate components: **bra_size** and **cup_size**. The **bust_size** variable combines both numerical (e.g., 34, 36) and alphabetical (e.g., A, B, C) parts. To make the data more usable for analysis, we extracted the numerical part into a new column called **bra_size** and the alphabetical part into another column named **cup_size**.

bra_size	cup_size
34.0	d
34.0	b
NaN	NaN
34.0	c
34.0	b

Fig 18: Feature Extraction

After the extraction, we found that some entries in **bust_size** were missing, which resulted in corresponding null values in the new columns. This transformation is needed because separating these components allows for a better analysis. It also ensures that numerical values can be used in quantitative analyses, while **cup_size** can be treated as a categorical variable for grouping and comparisons. This step simplifies the modeling process and makes the dataset more structured. [4]

4.2. Handling the Null Categorical Variable:

In this feature engineering stage, we focused on addressing missing values in our categorical variables. To keep things simple and consistent, we decided to replace all missing entries with the keyword "**missing**".

```
bust_size: 0.0956% missing values
rented_for: 0.0001% missing values
body_type: 0.076% missing values
cup_size: 0.0956% missing values
```

Fig 19: Missing Categorical Values

By assigning the label "**missing**" to these unknown categories, we ensure that our machine learning models can still process these records. It treats the missing information as its own category, which can be helpful if the absence of data carries some meaning.[4] For instance, if a customer chose not to provide their **body_type** or **rented_for** information, labeling it as "missing" lets the model recognize and learn from this pattern.

```
bust_size     0
rented_for    0
body_type     0
cup_size      0
dtype: int64
```

Fig 20: Handled Categorical Values

This method is straightforward and avoids the complexity of more advanced feature engineering techniques that might not be necessary for categorical data. It also prevents any bias that could occur if we tried to guess or fill in these missing values with the mean or some other value that might not accurately represent the missing data.

4.3 Handling the Null Numerical Variable:

We identified numerical features in the dataset that have missing values. These features include **weight**, **rating, height, age,** and **bra_size**, with varying percentages of missing data. To address these gaps, we chose to fill the missing values using the median value of each feature. The median is a better choice than the mean in this case because it is less affected by outliers, which are present in our dataset.

```
weight: 0.1557% missing value
rating: 0.0004% missing value
height: 0.0035% missing value
age: 0.005% missing value
bra_size: 0.0956% missing value
```

Fig 21: Missing Numerical Values

For each numerical feature with missing values, we calculated its **median** and replaced the missing entries with this value. Also, we created a new column for each feature to indicate whether a value was originally missing (marked as **1**) or not (marked as **0**). This new column helps us get information about the presence of missing values. This method ensures that we handle missing values without losing any rows from the dataset. It also reduces the impact of outliers and helps maintain the integrity of the numerical features for further analysis and modeling. [4]

```
weight      0
rating      0
height      0
age         0
bra_size    0
dtype: int64
```

Fig 22: Handled Numerical Values

5. Content-Based Recommender System

Now that all missing values have been handled, and the dataset is complete, we can proceed to build our first recommendation system: the content-based recommender system. This system focuses on suggesting items to users based on the attributes of the products they have previously used. To build the content-based recommender system, we started by selecting relevant features for recommendation. These include numerical attributes like **weight, height, size, age,** and **bra_size**, as well as categorical attributes like **rented_for, body_type, category, cup_size,** and **fit**. These features capture the key details of both the user and the product, ensuring personalized recommendations.

5.1. Data Preprocessing:

5.1.1. Feature Scaling:

Feature Scaling was applied to the numerical features, such as **weight, height, size, age, and bra_size**, using **StandardScaler**. Feature scaling makes sure that all numerical variables are on the same scale, which is critical for accurate similarity calculations in the recommendation process. Using **StandardScaler**, we transformed the numerical features into a standardized form where each feature has a mean of 0 and a standard deviation of 1. This was done by subtracting the mean and dividing by the standard deviation for each feature. After scaling, the numerical values are converted into comparable units.

5.1.2. Label Encoding:

Categorical variables like **rented_for, body_type, category, cup_size, and fit** were encoded using **LabelEncoding**. Encoding is necessary because machine learning algorithms and similarity calculations require numerical inputs, and categorical variables in their raw form (e.g., text values like "wedding" or "athletic") cannot be directly processed. Categorical features contain important details about items and users, such as why an item was rented (rented_for) or the user’s body type (body_type). These details are crucial for making accurate

recommendations. Encoding converts the text values (like "wedding" or "athletic") into numbers. The **encoded soup** combines all the important features—both numerical and categorical—into a single text-like format.

5.2. Model Development:

5.2.1. TF-IDF Vectorization:

TF-IDF (Term Frequency-Inverse Document Frequency) is a technique used to convert text data as numerical vectors. It evaluates the importance of each feature (term) in a **tokenized documented manner** to the entire dataset. In our case, it assigns weights to each component of the encoded_soup, giving more importance to unique or distinguishing features of an item. Once we created the encoded_soup, the next step was to convert it into a numerical format using TF-IDF Vectorization. The parameter ngram_range=(1,2) defines to the model that it needs to also vectorize sets of two words so that we can capture set of words such as 'Very Good' or 'Too Poor' etc.This step ensures that the recommender system can compare items numerically based on their attributes. TF-IDF divides the features into terms, calculates their frequency, and assigns a weight to each term based on its importance. Terms that are unique to an item are given higher importance, while commonly occurring terms receive lower weights.[5] We will later use this representation to find item similarity using an **Annoy index**, a tool optimized for efficient similarity searches.

```
tfidf = TfidfVectorizer(ngram_range=(1, 2))
tfidf_matrix = tfidf.fit_transform(dataset['encoded_soup'])
tfidf_matrix.shape
```

```
(192542, 5489)
```

Fig 23: TF-IDF

After generating the TF-IDF matrix, we converted it into a NumPy array using the .toarray() method. This step transforms the sparse matrix, which efficiently stores only non-zero values, into a dense array. The dense format is essential for performing similarity calculations, especially when using the Annoy index.

5.2.2. Annoy Index:

The next step in building our content-based recommender system is implementing the Approximate Nearest Neighbors Oh Yeah (ANNOY) Index, a sophisticated algorithm developed by Spotify for music recommendations and now widely used in recommendation systems. ANNOY addresses a fundamental challenge in large-scale recommendation systems: efficiently finding similar items in high-dimensional spaces without exhaustively comparing every pair of items. [6]

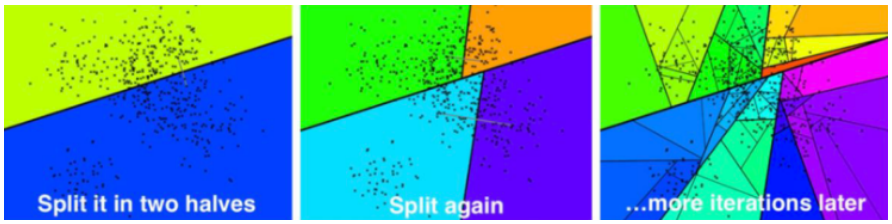

Fig 24: ANNOY algorithm visualized

To state the obvious, we did try implementing cosine similarity at first. With 192,544 reviews, computing pairwise similarities would require creating a matrix of size 192,544 x 192,544, demanding approximately 187 gigabytes of RAM. This computational limitation is a common challenge in recommendation systems with large datasets, as similarity calculations scale quadratically with the number of items. This explains why more memory-efficient methods like ANNOY (Approximate Nearest Neighbors) were necessary, as they avoid computing and storing the full similarity matrix while still maintaining good recommendation quality.

```
MemoryError: Unable to allocate 187. GiB for an array with shape (25057375706,) and data type int64
```

Fig 25: Cosine similarity error

ANNOY works by constructing multiple binary trees where each node represents a hyperplane that divides the space of item vectors into two subspaces. This division is performed recursively until reaching leaf nodes containing a small number of items. The key innovation lies in its approximation technique - instead of calculating exact distances between all item pairs (which would be computationally expensive with $O(n^2)$ complexity), ANNOY creates multiple

random projections of the data. This randomization, combined with the tree structure, allows for quick traversal to find nearest neighbors with O(log n) complexity. [7]

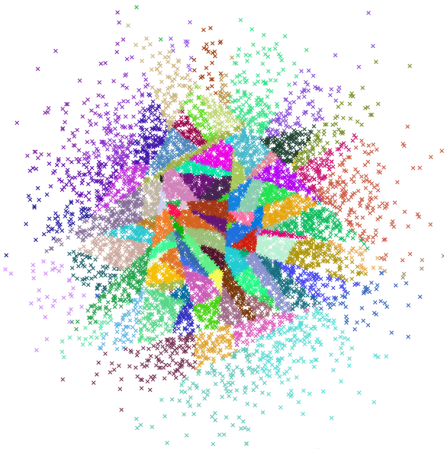

Fig 26: ANNOY cluster

In our clothing rental recommendation system, ANNOY utilizes the TF-IDF matrix we constructed from item descriptions, categories, and rental occasions. Each clothing item is represented as a high-dimensional vector in this space, where dimensions correspond to different terms and features. When searching for similar items, ANNOY traverses these binary trees, efficiently narrowing down the search space to find items with similar vector representations. This approach is particularly valuable for our system as it enables real-time recommendations even with our dataset of 5,850 unique items, maintaining both computational efficiency and recommendation quality.

The "approximate" nature of ANNOY makes a practical trade-off between accuracy and speed - while it might not always find the exact nearest neighbors, it consistently finds good approximations much faster than exhaustive search methods. This trade-off is controlled through parameters like the number of trees and search depth, allowing us to balance precision and computational resources based on our system's requirements.[6]

```
get_recommendations(annoy_index, 2260466)
```

	item_id	rented_for	body_type	category	rating
113113	268562	wedding	hourglass	gown	10.0
86493	1057309	everyday	Missing	dress	2.0
68571	1175903	formal affair	hourglass	gown	8.0
104180	963476	party	hourglass	sheath	10.0
65565	730008	wedding	straight & narrow	dress	10.0
119899	704861	everyday	petite	dress	6.0
19320	921642	wedding	Missing	dress	10.0
75337	986324	party	pear	dress	10.0
20814	127495	formal affair	petite	dress	4.0
39194	2495930	everyday	full bust	coat	10.0

Fig 27: Annoy Index

The output above represents the top 10 recommended items for a given item using the Annoy Index. These recommendations are based on the similarity of the encoded features of the query item (2260466) with other items in the dataset. The system has successfully identified items with similar attributes to the query item. For instance:

1. Items rented for weddings and formal occasions dominate the list, aligning with the likely context of the query item.
2. The diversity in body types and categories ensures that the recommendations are not overly narrow but still relevant to the query item's attributes.

5.3. Result Analysis:

5.3.1. Parameter Tuning:

Parameter tuning in our Annoy-based recommendation system primarily focused on optimizing the number of trees, a critical hyperparameter that balances accuracy and computational efficiency. While Annoy offers multiple tuning options, including search_k for query time optimization, our dataset size didn't warrant query speed adjustments. We conducted extensive experiments with three distinct tree configurations (100, 500, and 1000 trees) to identify the optimal trade-off between recommendation quality and computational resources.

This systematic evaluation helped us understand how the number of trees impacts the model's ability to capture item similarities in our clothing rental context.[6]

In this step, we evaluated the performance of our content-based recommender system using **5-fold cross-validation** and calculated metrics such as **Root Mean Squared Error (RMSE)** and **Mean Absolute Error (MAE).** These metrics measure how well the predicted ratings align with the actual ratings in the dataset, providing insights into the accuracy of the model.

Root Mean Squared Error (RMSE): Measures the average squared difference between predicted and actual ratings. Lower RMSE indicates better accuracy.

Mean Absolute Error (MAE): Measures the average absolute difference between predicted and actual ratings.

```
Average RMSE: 1.5637664856254951
Average MAE: 1.189229487991579
```

Fig 28: ANNOY performance for n_trees=100

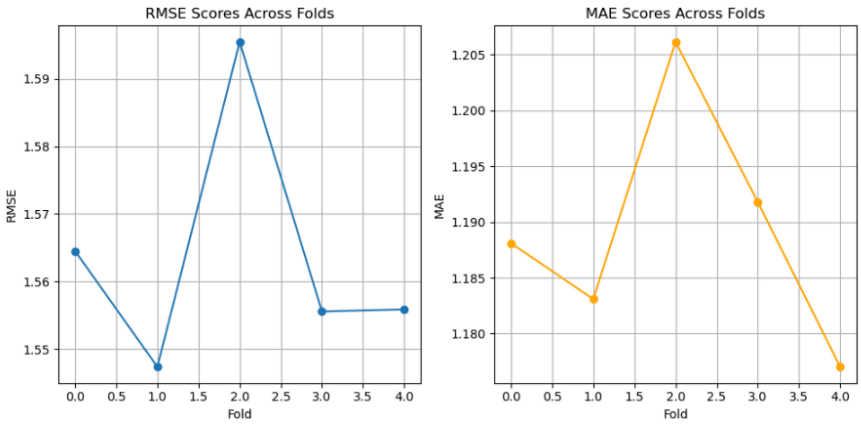

Fig 29: ANNOY cross validation for n_trees=100

In these images, we can see that the average root mean error square is 1.5673 while the mean error square for 100 trees in ANNOY index is 1.1892. Now, let us see the performance for n_trees=500

```
Average RMSE: 1.5652306193611136
Average MAE: 1.1902315414455567
```

Fig 30: ANNOY performance for n_trees=500

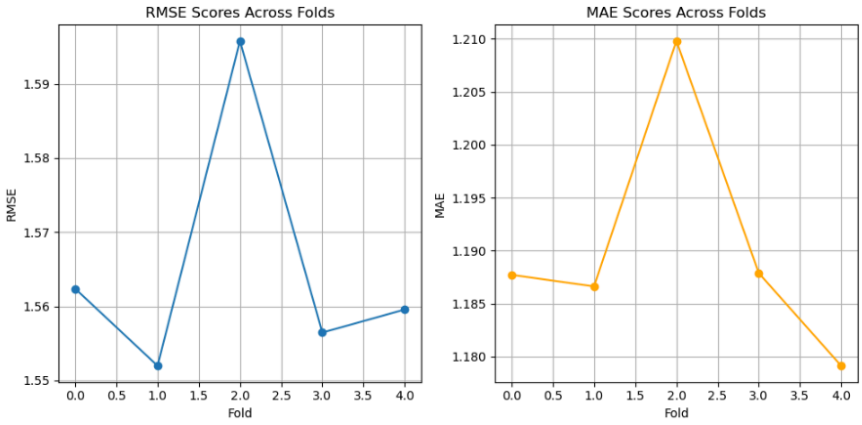

Fig 31: ANNOY cross validation for n_trees=500

In these images, we can see that the average root mean error square is 1.5652 while the mean error square for 500 trees in ANNOY index is 1.1902. We can see that it's not a major improvement from when it had 100 trees. Now, let us see the performance for n_trees=1000

```
Average RMSE: 1.5543210187150773
Average MAE: 1.1793106888849831
```

Fig 32: ANNOY performance for n_trees=1000

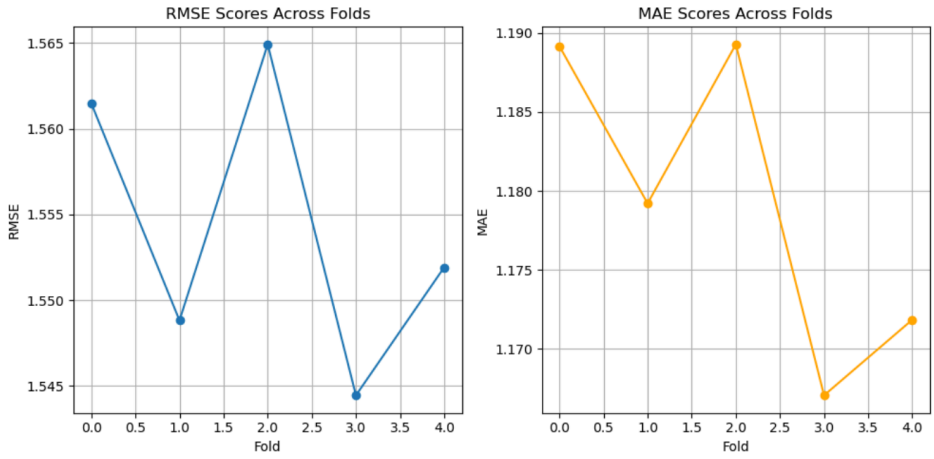

Fig 33: ANNOY cross validation for n_trees=1000

In these images, we can see that increasing the number of trees to 1000 does decrease the RMSE and MAE to some extent and hence we can say that n_trees=1000 yields slightly superior results. However, do note that the resulting training time increases quickly as the number of trees increases.

5.3.2. Performance Evaluation:

The model's final architecture incorporates a robust ensemble of 1,000 decision trees, complemented by sentiment logits derived from our deep learning analysis. This sophisticated combination allows us to capture both the nuanced patterns in customer preferences and the emotional context expressed in their reviews. The detailed sentiment analysis framework and its implications will be explored in subsequent sections.

```
Fold 1:
RMSE: 1.5580
MAE: 1.1801

Fold 2:
RMSE: 1.5458
MAE: 1.1842

Fold 3:
RMSE: 1.5864
MAE: 1.2025

Fold 4:
RMSE: 1.5546
MAE: 1.1762

Fold 5:
RMSE: 1.5485
MAE: 1.1778
```

Fig 34: RMSE and MAE Fold wise

```
Average RMSE: 1.5586470139294633
Average MAE: 1.1841347655325727
```

Fig 35: Avg RMSE and MAE

The evaluation of the content-based recommender system shows promising results. With an average RMSE of 1.558 and MAE of 1.184, the system demonstrates reasonably accurate predictions of user ratings. These metrics suggest that the model captures the key attributes influencing user preferences and makes recommendations that align closely with actual user behavior. The plots of these RMSE and MAE can be seen below :

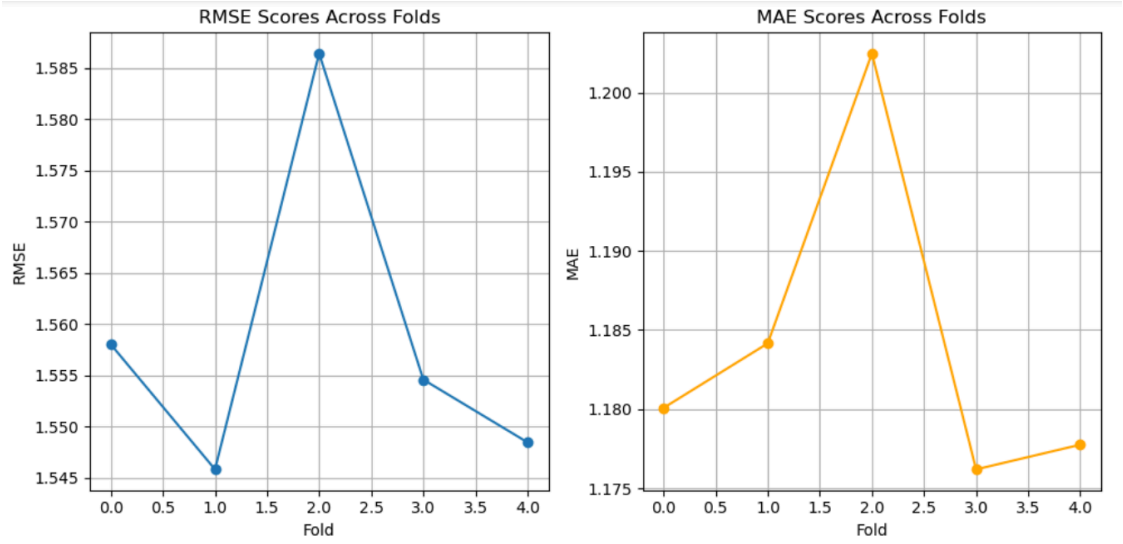

Fig 36: Content-based RMSE/MAE plots

6. Neural Graph Collaborative Filtering

6.1. GNN:

Graph Neural Networks (GNNs) represent a revolutionary advancement in deep learning, specifically designed to process and learn from interconnected data structures. Unlike traditional neural networks that operate on regular, grid-like data (such as pixels in images or sequences in text), GNNs excel at capturing complex relationships and patterns within irregular, graph-structured data where information flows through connected entities. [8]

At their core, GNNs work with two fundamental components: nodes (vertices) that represent entities, and edges that encode relationships between these entities. What makes GNNs particularly powerful is their ability to learn through message passing - a process where nodes iteratively exchange information with their neighbors, aggregating and transforming this information to create increasingly sophisticated representations of both individual nodes and their local network structure.

The true innovation of GNNs lies in their ability to maintain permutation invariance (the order of nodes doesn't matter) while capturing both local and global graph properties. Through multiple layers of message passing, GNNs can understand not just immediate connections but also higher-order relationships - how nodes influence each other through multiple steps of connectivity.[8] This hierarchical learning process mirrors how humans understand complex systems: by combining local patterns into increasingly abstract and comprehensive representations.

GNNs have transformed various fields where relationship modeling is crucial. In social networks, they can capture complex social dynamics; in chemistry, they model molecular structures; and in recommendation systems, they excel at understanding the intricate web of user-item interactions. Their success stems from their ability to leverage the inherent structure of graph data while maintaining end-to-end differentiability, allowing them to learn optimal representations directly from the data.

This architectural design makes GNNs particularly well-suited for tasks where understanding relationships and network effects is crucial for making accurate predictions or

recommendations. By learning to aggregate information from both immediate and distant connections, GNNs can capture subtle patterns and similarities that might be invisible to traditional machine learning approaches. [9]

6.2. Model Development:

6.2.1. Initializing the layer:

The first step is to set up the layer, which includes creating two mathematical tools (called W1 and W2) to transform user and item embeddings into new representations. These transformations allow the model to extract meaningful patterns from the data. To ensure the model learns effectively, a "Leaky ReLU" activation function is added. This function makes sure that no part of the model stops learning, even if some of the values during training are very small. Lastly, a technique called dropout is applied, which randomly turns off parts of the layer during training. This might sound counterintuitive, but it helps prevent the model from becoming overly reliant on certain patterns. This layer enables users and items to learn from the graph structure. For example, a user embedding updates based on the items they rented, and item embedding updates based on the users who interacted with them. Over time, the embeddings reflect direct and indirect relationships, improving recommendations. For instance, if similar users rented a handbag with a dress, the model can recommend the handbag, even if a user hasn't interacted with it directly.

```
class NGCFLayer(MessagePassing):
    def __init__(self, in_size, out_size, dropout=0.1):
        super().__init__(aggr='add')
        self.W1 = nn.Linear(in_size, out_size)
        self.W2 = nn.Linear(in_size, out_size)
        self.leaky_relu = nn.LeakyReLU(0.2)
        self.dropout = nn.Dropout(dropout) #Randomly "drops out" a portion o

    def forward(self, x, edge_index):
        row, col = edge_index
        deg = degree(row)
        deg_inv_sqrt = deg.pow(-0.5)
        deg_inv_sqrt[deg_inv_sqrt == float('inf')] = 0
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        out = self.propagate(edge_index, x=x, norm=norm)
        out = self.dropout(out)
        return self.leaky_relu(out + x)  # Apply activation after residual

    def message(self, x_j, norm):
        x_j = self.dropout(x_j)
        transformed = self.W1(x_j) + self.W2(x_j)
        return norm.view(-1, 1) * transformed
```

Fig 37: Collaborative Filtering Layer

6.2.2 The NGCF Model:

The NGCF model starts by setting up user and item embeddings, which represent users and items as numerical vectors in a shared high-dimensional space. These embeddings serve as the foundation for capturing user preferences and item characteristics. The embedding size is set to 64 dimensions by default, which balances computational efficiency. The embeddings are initialized with small random values using a normal distribution, which helps the model begin learning without introducing bias. The model also includes three NGCF layers stacked sequentially. Each layer performs message passing on the user-item graph, allowing nodes to learn from their neighbors. Stacking multiple layers enables the model to capture both direct and multi-hop relationships, where a node learns not only from its immediate connections but also indirectly through the graph structure. Finally, the model includes a prediction layer, implemented as a neural network with two linear layers. The first layer maps the combined user and item embeddings into a smaller space (32 dimensions), and the second layer predicts two outputs: a user's rating for an item (on a scale of 1 to 10) and the likelihood that the item fits the user. A ReLU activation function is applied between these layers to introduce non-linearity, enabling the network to capture complex patterns in the embeddings.

```
class NGCF(nn.Module):
    def __init__(self, n_users, n_items, emb_size=64):
        super().__init__()

        # Embeddings
        self.n_users = n_users
        self.n_items = n_items
        self.user_embedding = nn.Embedding(n_users, emb_size)
        self.item_embedding = nn.Embedding(n_items, emb_size)

        # NGCF layers
        self.layers = nn.ModuleList([
            NGCFLayer(emb_size, emb_size),
            NGCFLayer(emb_size, emb_size),
            NGCFLayer(emb_size, emb_size)
        ])

        # Prediction layers
        self.predictor = nn.Sequential(
            nn.Linear(emb_size*2, 32),
            nn.ReLU(),
            nn.Linear(32, 2)  # 2 outputs for rating and fit
        )
```

Fig 38: NGCF Model

6.2.3 The Forward Pass:

The model combines (or concatenates) all the user and item embeddings into a single data structure, which is then ready to be passed through the next layers of the model. Then the

combined embeddings are sent through a series of layers called NGCF layers. Each layer allows the embeddings to gather information from connected nodes in the graph.[8] For example, a user's embedding updates based on the items they interacted with, and an item's embedding updates based on the users who rented it. After each layer, the embeddings are normalized to keep the values stable and prevent any problems during training. The outputs from each layer are saved for later use. To make the most of the information learned in each layer, the model stacks all the embeddings from the different layers together and takes an average. This ensures that the final representation includes both "shallow" insights (direct relationships) and "deeper" insights (indirect relationships) from the graph. By doing this, the model creates a well-rounded understanding of users and items. After passing through the NGCF layers, the combined embeddings are split back into separate groups: one for users and one for items.

```
    # Get embeddings
    user_emb = self.user_embedding.weight
    item_emb = self.item_embedding.weight
    x = torch.cat([user_emb, item_emb])

    # Message passing
    all_embeddings = [x]
    for layer in self.layers:
        x = layer(x, edge_index) #Applies the current layer in loop to the em
        x = F.normalize(x) #Normalizes the embeddings after each layer. This
        all_embeddings.append(x)

    x = torch.stack(all_embeddings, dim=1)
    x = torch.mean(x, dim=1)

    # Split users and items
    users_emb = x[:self.n_users]
    items_emb = x[self.n_users:]

    # Get specific embeddings for prediction
    user_emb = users_emb[user_nodes]
    item_emb = items_emb[item_nodes]

    # Predict both rating and fit
    concat = torch.cat([user_emb, item_emb], dim=1)
    pred = self.predictor(concat)
    rating_pred = torch.sigmoid(pred[:, 0]) * 10.0  # Scale to 1-10 rating
# fit_pred = F.softmax(pred[:, 1].unsqueeze(-1), dim=1)  # Fit prediction

    return rating_pred
```

Fig 39: Ratings prediction using MessagePassing()

6.3. Result Analysis:

6.3.1. Parameter Training:

The optimization of our GNN-based recommendation system involved fine-tuning multiple hyperparameters to achieve optimal performance. These included architectural parameters (number of graph convolution layers, linear layer dimensions), training parameters

(learning rate, batch size, number of epochs), and model-specific parameters (activation functions, embedding dimensions). While an exhaustive exploration of the hyperparameter space would be beyond the scope of this report, we present a comparative analysis of the model's performance before and after optimization to demonstrate the effectiveness of our tuning process.

```
Final Cross-Validation Results:
Average RMSE: 1.6742 ± 0.0332
Average MAE: 1.4282 ± 0.0254
```

Fig 40: GNN's performance before parameters tuned

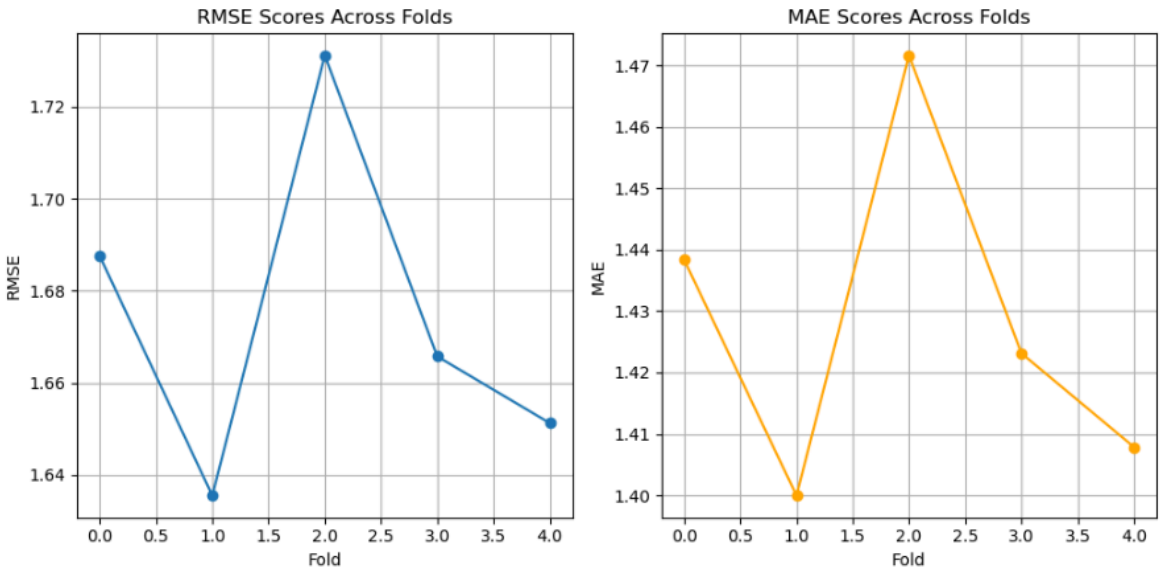

Fig 41: GNN's cross validation before parameters tuned

The initial model architecture featured two graph convolutional layers followed by a single linear prediction layer. Through iterative optimization, we enhanced the model by:

- Replacing standard ReLU with LeakyReLU activation functions to better handle negative values and prevent further problems
- Increasing the number of training epochs to allow for better convergence

- Reducing the batch size from 512 to achieve more stable gradient updates and improved model performance

These modifications significantly enhanced the model's ability to capture complex user-item relationships in our clothing rental recommendation system.

```
C:\Users\kisha\anaconda3\Lib\site-pacl
  warnings.warn(
Fold 1, Epoch 0, Loss: 15.6987
Fold 1, Epoch 2, Loss: 5.2217
Fold 1, Epoch 4, Loss: 1.9569
Fold 1, Epoch 6, Loss: 1.8062
Fold 1, Epoch 8, Loss: 1.6876
Fold 1 - RMSE: 1.5014, MAE: 1.2423
C:\Users\kisha\anaconda3\Lib\site-pacl
  warnings.warn(
Fold 2, Epoch 0, Loss: 14.8995
Fold 2, Epoch 2, Loss: 4.4153
Fold 2, Epoch 4, Loss: 1.9349
Fold 2, Epoch 6, Loss: 1.8110
Fold 2, Epoch 8, Loss: 1.6988
Fold 2 - RMSE: 1.4760, MAE: 1.2276
C:\Users\kisha\anaconda3\Lib\site-pacl
  warnings.warn(
Fold 3, Epoch 0, Loss: 16.0549
Fold 3, Epoch 2, Loss: 5.3382
Fold 3, Epoch 4, Loss: 1.9574
Fold 3, Epoch 6, Loss: 1.7990
Fold 3, Epoch 8, Loss: 1.6729
Fold 3 - RMSE: 1.5206, MAE: 1.2575
C:\Users\kisha\anaconda3\Lib\site-pacl
  warnings.warn(
Fold 4, Epoch 0, Loss: 18.1822
Fold 4, Epoch 2, Loss: 5.4548
Fold 4, Epoch 4, Loss: 1.9615
Fold 4, Epoch 6, Loss: 1.8080
Fold 4, Epoch 8, Loss: 1.6864
Fold 4 - RMSE: 1.5061, MAE: 1.2488
C:\Users\kisha\anaconda3\Lib\site-pacl
  warnings.warn(
Fold 5, Epoch 0, Loss: 20.1254
Fold 5, Epoch 2, Loss: 8.3801
Fold 5, Epoch 4, Loss: 2.3164
Fold 5, Epoch 6, Loss: 1.8243
Fold 5, Epoch 8, Loss: 1.6970
Fold 5 - RMSE: 1.5016, MAE: 1.2439

Final Cross-Validation Results:
Average RMSE: 1.5011 ± 0.0144
Average MAE: 1.2440 ± 0.0098
```

Fig 42: GNN's cross validation after parameters tuned

6.3.2. Training and Evaluating the Model:

The first step is to initialize encoders for users and items using LabelEncoder. This encoder assigns a unique numerical ID to each user and item in the dataset. For example, if there are 100 unique users, they will be assigned IDs ranging from 0 to 99. After encoding, we have encoded_user_ids and encoded_item_ids, which represent the transformed user and item IDs. The total number of unique users (n_users) and items (n_items) is calculated using the LabelEncoder. These numbers are essential for defining the structure of the embeddings in the model. Specifically, n_users tells the model how many unique user embeddings to create, and n_items tells it how many unique item embeddings are needed. The edge_index is created to represent the connections (or interactions) between users and items in the graph. In this context, the graph nodes are users and items, and the edges represent interactions such as ratings or rentals.

The edge index is constructed as a 2D tensor:

1. The first row contains the user IDs.
2. The second row contains the corresponding item IDs, adjusted by adding the total number of users (n_users). This ensures that user and item nodes have unique IDs within the graph.

By using array indices (np.arange) instead of direct values, the code creates an efficient mapping for users and items in the graph structure.

```
# Initialize encoders
user_encoder = LabelEncoder()
item_encoder = LabelEncoder()

# Encode IDs
encoded_user_ids = user_encoder.fit_transform(dataset['user_id'])
encoded_item_ids = item_encoder.fit_transform(dataset['item_id'])
```

```
# Create edge index
n_users = len(user_encoder.classes_)
n_items = len(item_encoder.classes_)
# edge_index = torch.tensor([
#     dataset['user_id'].values,
#     dataset['item_id'].values + n_users
# ], dtype=torch.long)

# Create edge index once
edge_index = torch.tensor([
    np.arange(len(user_ids)),  # Use array indices instead of values
    item_ids + n_users
], dtype=torch.long)
```

```
# Initialize model
model = NGCF(n_users, n_items)
```

Fig 43: Training the Model

We use K-Fold Cross-Validation to evaluate the model. This technique splits the dataset into five equally sized "folds," where the model is trained on four folds and tested on the remaining fold. The process is repeated five times, ensuring that each fold is used as the test set once. Stratification ensures that the distribution of ratings is preserved across folds, making the evaluation fair and consistent. Two empty lists, rmse_scores and mae_scores, are created to store the evaluation metrics for each fold: RMSE (Root Mean Squared Error): Measures how well the predicted ratings match the actual ratings, with more emphasis on large errors. MAE (Mean Absolute Error): Calculates the average of absolute differences between predicted and actual ratings, providing a straightforward measure of accuracy. These metrics will be computed for each fold and stored in the corresponding lists for comparison. Then, the user and item IDs are mapped to the graph using the edge_index, and the ratings are used to evaluate the model's predictions.

```
: # Initialize K-Fold with stratification
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Initialize score lists
rmse_scores = []
mae_scores = []

# Prepare data
user_ids = dataset['user_id'].values
item_ids = dataset['item_id'].values
ratings = dataset['rating'].values
```

Fig 44: Evaluating the Model

Through the cross-validation results, we observed that the model consistently achieves low error rates across all folds. This means the model successfully learns meaningful patterns in user-item interactions and applies this knowledge to make predictions for new scenarios. The steadily decreasing training loss in each fold shows that the model is effectively learning and refining the embeddings for users and items. For example, in Fold 1, the loss started at 15.6987 in the first epoch and reduced to 1.6876 by the final epoch, a clear indication of convergence. This pattern was observed across all folds, reflecting that our training approach—combining techniques like gradient clipping, regularization, and adaptive learning rate scheduling—allowed the model to train efficiently and avoid overfitting. The RMSE values, which range between 1.4760 and 1.5206, show that the model minimizes large prediction errors effectively. Similarly,

the MAE values, which fall between 1.2276 and 1.2575, indicate that the model's average error across all predictions is small and consistent. These results demonstrate that the NGCF model performs well in capturing the complex relationships between users and items in the graph structure. The final averaged results across all folds—an RMSE of 1.5011 with a standard deviation of 0.0144 and an MAE of 1.2440 with a standard deviation of 0.0098. The low standard deviations indicate that the model performs predictably, regardless of the specific training and testing splits. This consistency is a strong indicator that our model can generalize beyond the current dataset and would likely perform well on new, unseen data.

```
Final Cross-Validation Results:
Average RMSE: 1.5011 ± 0.0144
Average MAE: 1.2440 ± 0.0098
```

Fig 45: Output of the NGCF Model

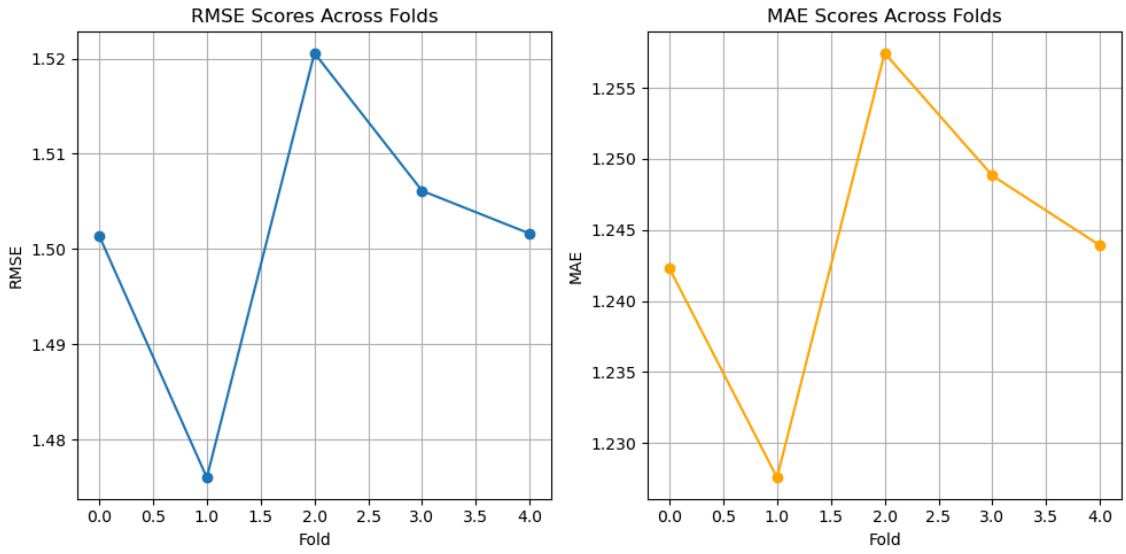

Fig 46: Neural Graph Collaborative Filtering - RMSE/MAE plots

6.4. Working of the NGCF Model:

The process begins by splitting the dataset into five folds, where the model is trained on four folds and tested on the remaining one. This ensures that every part of the data is used for testing at least once, providing a well-rounded evaluation of the model's performance. By leveraging cross-validation, the approach ensures that the model generalizes well across different data subsets, reducing the risk of overfitting to a particular fold. Within each fold, the user and item IDs are split into training and testing sets, and a corresponding edge index is created for the graph. This edge index connects users to items based on their interactions, enabling the NGCF model to learn user-item relationships effectively. The model is then initialized with the number of users, items, and embedding dimensions. It is optimized using the Adam optimizer, configured with a reduced learning rate for finer updates, and a scheduler is used to reduce the learning rate further when improvements stagnate. This setup ensures the model trains efficiently while avoiding overfitting or underfitting. The training process operates in batches, enabling the model to process large datasets efficiently. During each batch, the model predicts ratings for the given user-item pairs and calculates the loss using Mean Squared Error (MSE), a metric that penalizes large errors more heavily. To prevent overfitting, L2 regularization is applied to the model parameters. Gradient clipping is also used to limit the size of parameter updates, ensuring stable training even for large gradients. Additionally, the code includes a mechanism to skip batches with NaN loss values, preventing interruptions during training. Training continues for a maximum of 10 epochs, with early stopping in place to terminate training if no significant improvement is observed after five epochs. This prevents unnecessary computation and ensures the model does not overfit the training data. Once training for a fold is complete, the testing function (test_model) plays a critical role in evaluating the model on the test set. The function switches the model to evaluation mode, disabling training-specific operations like dropout. It uses the model to predict ratings for the test user-item pairs and ensures the predictions remain within the valid range (0 to 10) by clamping them. To handle any missing or invalid values in the predictions, these are replaced with the mean of the test ratings. The testing function ensures that the model's performance is quantified for each fold, and the results are stored for later analysis. After completing all folds, the results are aggregated to calculate the average RMSE and MAE across all folds, along with their standard deviations.

6.5. Bipartite Graph Structure:

The graph below represents a bipartite graph, specifically created to model the relationships between users and items in the dataset. In this visualization, nodes are divided into two distinct sets: users (represented by light blue circles) and items (clothes) (represented by light green circles). The graph is bipartite because edges only connect nodes from different sets—users are connected to items they interacted with, and no direct connections exist between two users or two items.

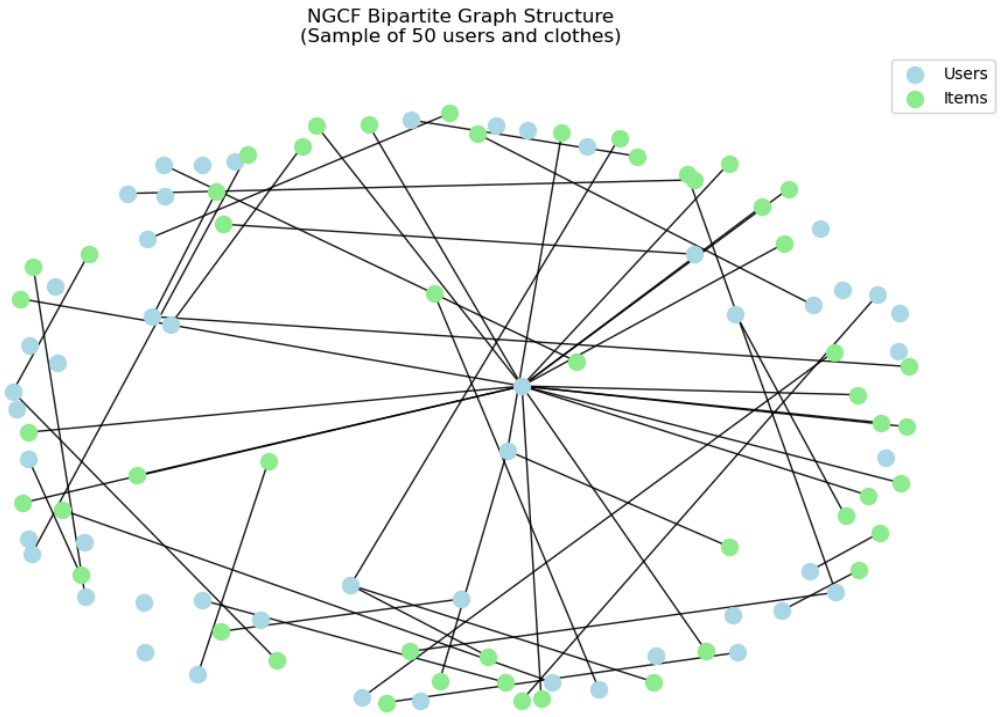

Fig 47: Bipartite Graph

This visualization uses a sample of 50 users and the items they interacted with, chosen from the dataset. The edges in the graph represent interactions, such as renting or reviewing an item. For example, if a light blue user node is connected to a light green item node, it indicates that the user rented or showed interest in that specific clothing item.

Looking at the structure of the graph, Dense Connections - some nodes (e.g., central items) have multiple edges connecting them to many users, indicating their popularity. These items likely appeal to a broad audience, making them important to consider for

recommendations. Sparse Connections - Nodes having fewer edges, suggesting that these items or users have more niche preferences or limited interactions.

This graph also reveals the diversity of interactions, Highly Connected Items - Items with many edges suggest high demand, such as popular dresses or accessories that many users rented. Individual User Patterns - Users with fewer connections may have more specific or unique preferences, which are also captured in this graph. [10]

6.6. Generating Recommendations:

In this final step, we implemented a function to generate personalized recommendations for a specific user based on the trained NGCF model. The goal of this process is to identify the top-rated items that the model predicts the user would most likely prefer. The personalized recommendation system ensures that each user receives suggestions tailored to their unique preferences and past interactions.

Encoding the User ID: The function begins by converting the provided user ID into its encoded form using the **LabelEncoder** we created earlier. This ensures that the user ID is mapped correctly to the internal representation used by the model.

Creating Input Data for Predictions: The function generates a list of all possible items available in the dataset. For each item, a temporary "edge index" is created, simulating a graph where the given user is connected to every item. This allows the model to predict how much the user might like each item, even if the user has not interacted with some of these items before.

Generating Predictions: Using the NGCF model, the function predicts a rating for every possible item for the user. These predictions represent how likely the user is to prefer each item, with higher scores indicating stronger preferences.

Selecting the Top-N Recommendations: The predicted ratings are sorted in descending order to identify the items with the highest scores. From this sorted list, the top-N items are selected (in this case, 10 recommendations). These items are expected to align most closely with the user’s preferences.

Mapping Back to Original Data: The encoded item IDs for the top recommendations are converted back to their original IDs using the LabelEncoder. The function then retrieves additional information about each recommended item, such as its category, the context for which it was rented (e.g., wedding or formal affair), and the actual user rating (if available).

	item_id	category	rented_for	rating	predicted_rating
36	1064397	gown	wedding	8.0	9.280361
14	123793	gown	formal affair	10.0	9.278329
1255	709832	gown	formal affair	10.0	9.274488
89	714374	dress	formal affair	10.0	9.271017
319	1213427	gown	formal affair	10.0	9.263975
1751	1260666	dress	date	8.0	9.255869
3434	304354	dress	work	8.0	9.246832
4813	1003076	dress	wedding	10.0	9.245588
2931	903647	gown	wedding	10.0	9.233828
986	724319	dress	wedding	10.0	9.231696

Fig 48: Top 10 Recommendations

The table displayed shows the output for a specific user. It includes recommendations for various gowns and dresses, tailored to different purposes such as weddings, formal affairs, or work. For instance, the first recommendation is a gown rented for a wedding, with an actual user rating of 8.0 and a predicted rating of 9.28. This indicates that the model has successfully predicted a high preference for this item, even higher than the original rating. The predicted ratings are consistent and close to the user’s actual preferences, demonstrating the model's accuracy in understanding user behavior. For example, several dresses recommended for weddings and formal affairs received predicted ratings close to 10, aligning well with their high actual ratings.

7. Sentiment Analysis

7.1. The Sentiments:

The sentiment analysis model we built works by combining a powerful pre-trained language model with custom rules to fine-tune its predictions. The foundation of the model is **textattack/bert-base-uncased-SST-2** [11], a version of BERT (Bidirectional Encoder Representations from Transformers) trained specifically for classifying text sentiment. BERT is an advanced machine learning model that understands the context of words in a sentence, making it great at interpreting whether a review is positive or negative. However, BERT alone might not always capture the nuances of language specific to our dataset, so we added some customizations to improve its performance.

The process starts by cleaning and preparing the dataset. Each review is checked to ensure it contains valid text, and we remove any empty or incomplete entries. These reviews are then passed to a tokenizer, which breaks them down into smaller pieces (called tokens) that the BERT model can understand. The tokenizer also ensures that all reviews are the same length by either truncating long reviews or padding shorter ones with placeholders. This preparation ensures that every review is in a format compatible with the BERT model. [12]

```
# Tokenizer and model initialization
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertForSequenceClassification.from_pretrained("textattack/bert-base-uncased-SST-2")
```

Fig 49: Tokenizer

Once the reviews are tokenized, they are fed into the BERT model, which generates initial predictions. BERT does this by analyzing how words in the review relate to one another

and to the overall meaning of the text. For example, it can distinguish between a sentence like "The dress is great" (positive) and "The dress isn't great" (negative) because it understands how the word "isn't" changes the sentiment. At this stage, the model outputs a score for each review, indicating how likely it is to be positive or negative.

After this initial step, we refine the predictions using two custom adjustments: keyword analysis and review length. First, we created lists of positive and negative keywords that people often use in reviews. Positive keywords include words like "perfect," "comfortable," and "amazing," while negative keywords include words like "tight," "uncomfortable," and "poor." If a review contains many positive keywords, its positive sentiment score is increased. Conversely, reviews with negative keywords see their negative sentiment score go up. This adjustment helps the model align more closely with patterns in our dataset.

```
# Define keywords and their weights
positive_keywords = [
    'great', 'perfect', 'loved', 'comfortable', 'compliments', 'fantastic', 'excellent', 'beautiful', 'flattering',
    'love', 'perfectly', 'nice', 'cute', 'good', 'gorgeous', 'amazing', 'liked', 'fine', 'loved dress', 'fit perfectly',
    'received compliments', 'got compliments', 'dress fit', 'great dress', 'dress perfect', 'dress beautiful',
    'compliments night', 'beautiful dress', 'highly recommend', 'lots compliments', 'form fitting', 'fit perfect', 'fun',
    'sexy', 'worked', 'snug', 'tailored', 'sleek', 'elegant', 'chic', 'stylish', 'form-fitting', 'seamless', 'cozy', 'luxurious',
    'well-made', 'high-quality', 'durable', 'breathable', 'soft', 'supple', 'flexible', 'adaptable', 'polished', 'refined',
    'trendy', 'modern', 'classic', 'sophisticated', 'smart', 'neat', 'trim', 'crisp', 'fitted', 'customized', 'bespoke',
    'adjustable', 'supportive', 'lightweight', 'airy', 'smooth', 'silky', 'plush', 'resilient', 'robust', 'sturdy', 'reliable',
    'dependable', 'practical', 'functional', 'effortless', 'easy-to-wear', 'tailored to perfection', 'fits like a dream',
    'like a second skin', 'ultimate relaxation', 'moves with you', 'perfectly cozy'
]

negative_keywords = [
    'little', 'small', 'tight', 'didnt', 'runs', 'big', 'long', 'back', 'short', 'waist', 'chest', 'wasnt', 'hips',
    'backup', 'smaller', 'bigger', 'cut', 'loose', 'larger', 'sizing', 'runs small', 'little big', 'runs large', 'little tight',
    'ordered size', 'little large', 'size size', 'backup size', 'definitely runs', 'bit large', 'recommend sizing',
    'dress ran', 'bit small', 'fit little', 'bit big', 'larger size', 'fashion tape', 'bust area',
    'rib cage', 'runsmall', 'run big', 'runs bit', 'dress runs small', 'size smaller', 'size bigger', 'size backup',
    'little short', 'dress tight', 'tight dress', 'however', 'unfortunately', 'wouldnt', 'ended wearing', 'end wearing', 'just didnt',
    'didn wear', 'wasn flattering', 'wouldn rent', 'didn love', 'unable wear', 'ran small', 'ran big', 'customer service',
    'self conscious', 'terrible', 'poor', 'bad', 'uncomfortable', 'horrible', 'ill-fitting', 'tight', 'loose', 'awkward',
    'restrictive', 'baggy', 'saggy', 'misfitting', 'oversized', 'undersized', 'constricting', 'sloppy', 'shapeless', 'unflattering',
    'bulky', 'droopy', 'hanging', 'tent-like', 'billowy', 'too big', 'too loose', 'too tight', 'pinching', 'squeezing', 'binding',
    'unbreathable', 'cramped', 'stiff', 'immobile', 'itchy', 'scratchy', 'irritating', 'unbearable', 'discomfort', 'annoying',
    'displeasing', 'disappointing', 'unsatisfactory', 'dissatisfied', 'unappealing', 'unattractive', 'unpleasant', 'embarrassing',
    'self-conscious', 'awkward', 'distressing', 'disheartening', 'off-putting', 'exasperating'
]

# Define weights for keyword occurrences and review length
keyword_weight = 2   # Weight added for each keyword occurrence
length_factor = 0.025   # Weight for review length
```

Fig 50: The Keywords

7.2. Fine-tune Prediction function:

The predict_sentiments function is designed to classify the sentiment of customer reviews as positive or negative. It takes three main inputs: the **dataloader** containing batches of

tokenized reviews, the model (a pre-trained BERT model for sentiment classification), and the df, which is the original dataset of reviews. Let's break down how the function works step by step.

The first step is to set the model into evaluation mode using model.eval(). This tells the model that it's being used for inference, not training, which disables features like dropout to ensure consistent predictions. Two empty lists, logits_list and predictions, are initialized to store the adjusted sentiment scores (logits) and final predictions for each review. Next, we use a torch.no_grad() block to prevent unnecessary calculations for gradients, which reduces memory usage and speeds up the process during inference. Inside this block, the function processes each batch of reviews from the data loader. For every batch, the input_ids and attention_mask are extracted. These represent the tokenized review text and the attention information that helps BERT focus on relevant parts of the input.

```
def predict_sentiments(dataloader, model, df):
    model.eval()
    logits_list = []
    predictions = []

    with torch.no_grad():
        for idx, batch in enumerate(dataloader):
            input_ids = batch["input_ids"]
            attention_mask = batch["attention_mask"]
```

Fig 51: Classify Sentiments

The tokenized data is passed into the BERT model, which generates raw logits for each review. Logits are essentially the unprocessed scores that the model uses to determine the likelihood of each sentiment class (positive or negative). These logits are then moved to the CPU and converted into a NumPy array for further adjustments.

For each review in the batch, the function retrieves the original review text from the dataset (df) based on its index. It converts the review text to lowercase to standardize it for keyword analysis. The length of the review (in words) is also calculated at this stage.

The function then adjusts the logits based on two custom rules: Keyword Adjustment: It checks how many positive and negative keywords appear in the review. For every occurrence of a positive keyword, the positive sentiment score (logit[1]) is increased by a predefined weight (keyword_weight). Similarly, for every negative keyword, the negative sentiment score (logit[0]) is increased. This adjustment helps the model account for the presence of words or phrases that strongly indicate sentiment.

Length Adjustment: The function amplifies the dominant sentiment score (either positive or negative) based on the length of the review. A longer review contributes more to its dominant sentiment because it's likely to provide more context. The adjustment is calculated as the length of the review multiplied by a small scaling factor (length_factor). If the review is more positive, the adjustment is added; if it's more negative, the adjustment is subtracted. After these adjustments, the logits are stored in logits_list, and the final sentiment prediction is determined using np.argmax(logit). This function selects the class (positive or negative) with the higher adjusted score, and the result is appended to the predictions list. When all batches are processed, the function returns two results, logits_list: A list of the adjusted logits for each review, showing the final scores for both positive and negative classes. predictions: A list of numerical labels (0 for negative, 1 for positive) representing the predicted sentiment for each review.

7.3. Model Evaluation:

To evaluate how well our sentiment analysis model performs, we compared its predictions with the true sentiment of the reviews. The true sentiments were determined based on the review ratings, with reviews above the average rating labeled as positive (1) and those below it as negative (0). By calculating various metrics, we gained insight into the model's accuracy, precision, recall, and overall performance. The accuracy of the model is 70%, meaning that 7 out of 10 reviews were classified correctly. This provides a general sense of how well the model is performing overall. However, accuracy alone doesn't tell the full story, especially when the dataset is imbalanced (e.g., more positive reviews than negative ones), so we also looked at other metrics. The precision for the positive class is 74%, which means that when the model predicts a review is positive, it is correct 74% of the time. This shows the model is good at minimizing false positives for positive reviews.

```
Performance Metrics:
Accuracy: 0.70
Precision: 0.74
Recall: 0.83
F1-Score: 0.78

Detailed Classification Report:
              precision    recall  f1-score   support

    Negative       0.60      0.46      0.52     67923
    Positive       0.74      0.83      0.78    124619

    accuracy                           0.70    192542
   macro avg       0.67      0.65      0.65    192542
weighted avg       0.69      0.70      0.69    192542
```

Fig 52: Performance Metrics

The recall for the positive class is 83%, showing that the model successfully identifies 83% of all actual positive reviews. Recall reflects how well the model catches all instances of a specific class. The F1-score balances precision and recall. For positive reviews, the F1-score is 78%, indicating strong performance in identifying positive sentiment. For negative reviews, the F1-score is 52%. The macro average of precision, recall, and F1-score treats both classes equally and shows lower values (precision: 67%, recall: 65%, F1-score: 65%) due to the poorer performance on negative reviews. The weighted average, which accounts for the larger number of positive reviews, reflects better overall scores, aligning closer to the metrics for positive sentiment (accuracy: 70%, precision: 69%, recall: 70%).

We leveraged Python's WordCloud library to perform visual text analysis of customer reviews in our clothing rental dataset. This visualization technique helped identify frequently occurring terms in different sentiment categories. By generating separate word clouds for positive and negative reviews, we uncovered distinct vocabulary patterns that customers use when expressing satisfaction versus dissatisfaction with their rental experiences.[13] This analysis provided valuable insights into the specific aspects of clothing items and rental services that drive customer sentiment, helping understand what features lead to positive experiences and what issues commonly cause customer concerns.

Fig 53: Wordcloud of overall review text

Fig 54: Wordcloud of positive review text

The prominence of words like "perfect," "loved," and "comfortable" indicates high customer satisfaction. Terms related to fit, compliments, and fabric quality dominate, suggesting customers particularly value well-fitting, aesthetically pleasing rentals that receive social validation.

Fig 55: Wordcloud of negative review text

Size-related concerns dominate with words like "tight," "big," and "small" appearing frequently. Issues with specific garment areas ("waist," "back," "chest") and fit problems are common pain points, indicating sizing accuracy is a crucial factor in customer dissatisfaction.

8. Graph-Based Hybrid Recommender Analyzing Sentiment Patterns

8.1. Hybrid formulization of rating prediction:

The hybrid_recommendations function is designed to provide personalized recommendations to users by combining two approaches: NGCF (Neural Graph Collaborative Filtering) and content-based filtering. [14] This hybrid method balances recommendations based on user interactions (NGCF) with recommendations that focus on the similarity between items (content-based filtering). By blending these approaches, the system ensures diverse and accurate recommendations tailored to the user's preferences.

Our hybrid recommendation system addresses one of the fundamental challenges in recommender systems - the cold start problem, where new users have no rental history or new items have no interactions. This problem typically limits the effectiveness of purely collaborative filtering approaches, as they rely solely on historical interaction patterns. [15]

The system combines the strengths of both collaborative (NGCF) and content-based filtering through a carefully orchestrated process. It begins by generating NGCF recommendations, leveraging the graph neural network's ability to capture complex user-item interaction patterns and higher-order relationships in the rental history. For users with existing rental history, the system also employs content-based filtering using Annoy index to identify items similar to their past preferences based on features like category and item descriptions.

The recommendations from both methods undergo score normalization to a 0-1 scale, ensuring fair contribution from each approach. The final hybrid score is computed using weighted averaging, with NGCF receiving a weight of 0.65 and content-based recommendations 0.35. This weighting scheme was chosen because:

1. NGCF captures dynamic user preferences and community wisdom (65% weight)
2. Content-based filtering provides stability and handles cold start scenarios (35% weight)
3. The ratio balances personalization with feature-based similarity

When faced with new users or items, the system can fall back on content-based recommendations, ensuring meaningful suggestions even without historical data. This hybrid approach ensures robust recommendations that adapt to both user history availability and item similarity patterns in our clothing rental context.

8.2. Hybrid Score:

The system generated personalized recommendations for the user with ID 420272 using a hybrid recommendation model. This approach combines predictions from NGCF (Neural Graph Collaborative Filtering) and content-based filtering, ensuring a balance between recommendations based on user interactions and those derived from item similarity. The results are ranked by their hybrid score, which reflects how well each item aligns with the user's preferences, based on both methods.

	item_id	category	rented_for	hybrid_score
49	1662825	sheath	party	0.650000
35	1076484	dress	formal affair	0.593310
10	148089	dress	wedding	0.592441
17	221704	dress	other	0.587112
61	2396986	coat	everyday	0.360546
1	123793	gown	wedding	0.350000
2	126335	dress	party	0.350000
3	127865	gown	formal affair	0.350000
5	139086	gown	formal affair	0.350000
7	144714	gown	wedding	0.350000

Fig 37: Recommendations based on Hybrid_Score

Collaborative filtering, as implemented through NGCF (Neural Graph Collaborative Filtering), focuses on analyzing the relationships between users and items. It looks at patterns in user behavior—such as what items a user has interacted with—and compares these patterns to

other users. For example, if User A and User B have interacted with similar items, NGCF can recommend additional items that User B liked to User A, even if those items are completely unfamiliar to User A. This method is particularly useful for discovering new items that a user might like, based on the preferences of similar users. However, collaborative filtering has a limitation known as the cold start problem, which occurs when there isn't enough data about a user or item. For instance, a new user without prior interactions or a new item without user feedback cannot be effectively included in recommendations. This is where the content-based filtering component of the hybrid model becomes valuable.

Content-based filtering focuses on the features of the items themselves, such as categories, descriptions, or metadata, to find similar items. For example, if a user has previously interacted with a "blue evening gown," content-based filtering can recommend other gowns with similar features, such as similar colors, styles, or categories. This method doesn't depend on other users' behavior, making it a perfect complement to collaborative filtering, especially for cold start scenarios or when specific user-item interactions are limited.

8.3. Comparison between Recommendations:

The comparison process works by analyzing specific attributes of both the user and the recommended item to determine how well they align. The goal is to ensure that the recommendation is not only relevant to the user's preferences but also practical in terms of fit, style, and intended use. This is achieved by retrieving and comparing data for key attributes like size, occasion, body type, and other relevant factors.

```
rented_for     vacation
age                28.0
size                 14
bust_size           34d
category         romper
rating             10.0
body_type     hourglass
height           172.72
weight            137.0
Name: 0, dtype: object
```

```
print(dataset[dataset['item_id']
```

```
rented_for        party
age                43.0
size                 14
bust_size           36d
category         sheath
rating             10.0
body_type     hourglass
height           162.56
weight            141.0
Name: 3347, dtype: object
```

Fig 38: Model Evaluation

The first step involves retrieving the user's attributes from the dataset. These attributes include information about the user's previous rentals, physical characteristics, and preferences. For example, the system looks at the user's age, size, bust size, body type, height, and weight, as well as details about their most recent rental, such as the occasion they rented for (rented_for) and the category of the item they chose. This profile serves as the basis for understanding the user's typical preferences and requirements. Next, the system retrieves the attributes of the recommended item. Similar to the user profile, the item's profile contains information about its size, category, and suitability for specific body types or occasions. For example, the recommended item might be described as ideal for a "party" and associated with a particular body type, height, or weight range. Additionally, the system looks at user ratings for the item to assess its overall appeal and satisfaction level among similar users. Once both sets of attributes are collected, the system performs a side-by-side comparison. It evaluates how closely the user's attributes match the characteristics of the item. For example: If the user's body type is "hourglass" and the item is designed for an "hourglass" figure, this increases the likelihood of a good fit. If the user's preferred size is 14 and the item is available in size 14, this ensures compatibility. The user's most recent rental occasion (e.g., "vacation") is compared to the recommended item's intended use (e.g., "party") to check if the recommendation diversifies or aligns with the user's needs. After comparing attributes, the system identifies areas of strong

alignment and potential mismatches. For instance, if the user's height and weight are close to the item's ideal measurements, it signals a high likelihood of fit. However, if the recommended item is associated with an older demographic or a different occasion, the system interprets this as an attempt to introduce variety while still ensuring relevance.

8.4. Model Evaluation:

The evaluation process is designed to measure how effectively the recommendation system aligns its suggestions with a user's preferences and attributes. This process begins by retrieving the user's data from the dataset, which includes detailed information about their most recent rental. For example, the system looks at the category of the clothing they rented, such as a "romper" or a "gown," the occasion it was rented for, such as a "vacation" or a "party," and the user's body type, like "hourglass" or "athletic." These attributes create a baseline profile that the system uses to assess whether the recommendations match what the user might find appealing or suitable. To simplify the evaluation process, the system uses predefined groupings for clothing categories and occasions. This grouping allows for broader comparisons when exact matches between user data and recommendations might not exist. For instance, items like "gown" and "dress" are grouped under "formal wear," while events like "wedding" or "formal affair" are categorized together. This enables the system to align recommendations at a more general level, ensuring flexibility when assessing matches. The recommendation system generates two sets of suggestions for the user. The first comes from the GNN model, which uses collaborative filtering to predict items based on patterns of user interactions. This method analyzes user behavior, such as past rentals and preferences, and suggests items that similar users have liked. The second set of recommendations comes from the content-based filtering model, which focuses on the characteristics of items the user has interacted with, finding others with similar features. Together, these two approaches provide a comprehensive list of potential recommendations [16]. Each recommendation is then evaluated against the user's profile to determine how closely it matches their preferences. This evaluation focuses on three main attributes. The system checks whether the recommended item belongs to the same general category as the user's past rental, such as "formal wear" or "casual tops." It also assesses whether the recommendation aligns with the occasion the user rented for, such as a "party" or a "vacation." Lastly, the system looks at

whether the recommended item is associated with the same body type as the user, such as “hourglass” or “pear.”

```
Average Matches Across All Users:
Category Match: 81.00%
Event Match: 56.00%
Body Type Match: 17.00%
```

Fig 39: System Performance

This process is repeated for every user in the dataset, and the results are aggregated to determine how well the system performs overall. The evaluation shows that the models perform strongly in matching recommendations to users’ preferred clothing categories, with an average category match of 81%. However, the results also reveal that recommendations align with users’ intended occasions only 56% of the time, and body type matches occur just 17% of the time. By analyzing the system’s performance across these attributes, the evaluation provides a clear picture of its strengths. It demonstrates how the recommendation system successfully captures broad patterns, such as clothing style, while highlighting areas like event-specific matching and body type alignment that could benefit from refinement. This evaluation process ensures that the system continues to improve in providing personalized and relevant recommendations for users.

9. Discussion

In conclusion, our GRASP (Graph-based Hybrid Recommender Analyzing Sentiment Patterns) system demonstrates robust performance across multiple evaluation metrics. The system achieved an RMSE of 1.5011 and MAE of 1.2440, indicating strong predictive performance. The hybrid approach, weighting NGCF at 65% and content-based filtering at 35%, successfully addresses the cold-start problem while maintaining recommendation quality.

Customer satisfaction metrics further validate the system's effectiveness. With a Net Promoter Score (NPS) of 42.70, the platform shows strong customer loyalty, where 71.06% of users are promoters, 27.75% are detractors, and only 1.20% remain passive. The Customer Satisfaction Score (CSAT) of 71.64% indicates that nearly three-quarters of users are satisfied with their rental experiences. These metrics align with the sentiment analysis results, where positive reviews frequently mention terms like "perfect," "loved," and "comfortable."

```
NPS Distribution:
nps_category
Promoter     71.056185
Detractor    27.748232
Passive       1.195583
Name: proportion, dtype: float64
```

Fig 39: NPS Distribution

```
Net Promoter Score (NPS): 42.70
Customer Satisfaction Score (CSAT): 71.64%
```

Fig 40: NPS and CSAT Scores

Similar graph-based hybrid systems have shown promise in various domains. Pinterest's PinSage uses GraphSAGE for content recommendations [18], while Amazon employs a

dual-embedding approach for product suggestions. However, our system's unique integration of sentiment analysis with graph structure sets it apart, particularly in the fashion rental context where fit and style preferences are crucial.

The evaluation metrics reveal that while our system excels at matching clothing categories (81% accuracy), there's room for improvement in occasion-specific (56%) and body type (17%) recommendations. The high promoter percentage (71.06%) suggests that users particularly value the system's ability to provide personalized recommendations that match their style preferences. Future enhancements could include incorporating more sophisticated attention mechanisms or implementing dynamic weighting schemes that adjust based on user interaction patterns.

A promising direction for future work would be integrating Spotify's Voyager, which has shown remarkable improvements over traditional Annoy-based approaches. Voyager offers 10 times faster speed at the same recall level and up to 50% better accuracy at equivalent speeds [17]. Additionally, its 4 times lower memory usage and ability to handle real-time updates make it particularly suitable for dynamic fashion rental platforms. Unlike Annoy, Voyager can utilize existing vectors within its index for searching similar items, potentially enhancing our recommendation quality while maintaining computational efficiency.

The success of our hybrid approach underscores the potential of graph-based architectures in recommendation systems, particularly when combined with sentiment analysis for nuanced understanding of user preferences. As the fashion rental industry continues to grow, such sophisticated recommendation systems will become increasingly vital for enhancing user experience and driving business success.

10. References

[1] How Rent The Runway Created a Multi-million Dollar Legacy. (n.d.). Www.referralcandy.com. https://www.referralcandy.com/blog/rent-the-runway-marketing-strategy

[2] Shin, E., Chung, T.-L. D., & Damhorst, M. L. (2022). Developing a scale to measure problems in finding a good fit. *Journal of Fashion Marketing and Management: An International Journal*, 1–17. https://doi.org/10.1108/jfmm-11-2021-0302

[3] Content-Based Recommendation System using Word Embeddings. (n.d.). KDnuggets. https://www.kdnuggets.com/2020/08/content-based-recommendation-system-word-embeddings.html

[4] Kuhn, M., & Johnson, K. (2019). Feature Engineering and Selection. Chapman and Hall/CRC. https://doi.org/10.1201/9781315108230

[5] How to perform text-based recommendations using TF-IDF. (2015). Educative. https://www.educative.io/answers/how-to-perform-text-based-recommendations-using-tf-idf

[6] Approximate Nearest Neighbors Oh Yeah (Annoy) - Zilliz blog. (2024). Zilliz.com. https://zilliz.com/learn/approximate-nearest-neighbor-oh-yeah-ANNOY

[7] Bernhardsson, E. (n.d.). Nearest neighbors and vector models – part 2 – algorithms and data structures. Erik Bernhardsson. https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html

[8] Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., & Sun, M. (2020). Graph neural networks: A review of methods and applications. AI Open, 1, 57–81. https://doi.org/10.1016/j.aiopen.2021.01.001

[9] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2020). A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, *32*(1), 1–21. https://doi.org/10.1109/TNNLS.2020.2978386

[10] Liu, M., Gao, H., & Ji, S. (2020). Towards Deeper Graph Neural Networks. Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. https://doi.org/10.1145/3394486.3403076

[11] Morris, J. X., Lifland, E., Yoo, J. Y., Grigsby, J., Jin, D., & Qi, Y. (2020). TextAttack: A Framework for Adversarial Attacks, Data Augmentation, and Adversarial Training in NLP. ArXiv:2005.05909 [Cs]. https://arxiv.org/abs/2005.05909

[12] EDA and Preprocessing for BERT. (n.d.). Kaggle.com. https://www.kaggle.com/code/parulpandey/eda-and-preprocessing-for-bert

[13] Chapter 32 Sentiment analysis and wordcloud | Spring 2021 EDAV Community Contributions. (n.d.). In jtr13.github.io. https://jtr13.github.io/cc21/sentiment-analysis-and-wordcloud.html

[14] Channarong, C., Paosirikul, C., Maneeroj, S., & Takasu, A. (2022). HybridBERT4Rec: a hybrid (content-based filtering and collaborative filtering) recommender system based on BERT. IEEE Access, 1–1. https://doi.org/10.1109/access.2022.3177610

[15] Singh, G. (2024, March 12). Solving the Cold Start Problem in Collaborative Recommender Systems. Tredence. https://www.tredence.com/blog/solving-the-cold-start-problem-in-collaborative-recommender-systems

[16] Guo, Q., Zhuang, F., Qin, C., Zhu, H., Xie, X., Xiong, H., & He, Q. (2020). A Survey on Knowledge Graph-Based Recommender Systems. IEEE Transactions on Knowledge and Data Engineering, 1–1. https://doi.org/10.1109/tkde.2020.3028705

[17] Engineering, S. (2023, October 25). Introducing Voyager: Spotify's New Nearest-Neighbor Search Library. Spotify Engineering. https://engineering.atspotify.com/2023/10/introducing-voyager-spotifys-new-nearest-neighbor-search-library/

[18] Agarwal, P., SK, M. I., Pancha, N., Hazra, K. S., Xu, J., & Rosenberg, C. (2024). OmniSearchSage: Multi-Task Multi-Entity Embeddings for Pinterest Search. Companion Proceedings of the ACM Web Conference 2024, 121–130. https://doi.org/10.1145/3589335.3648309