Wilson Sugiarto, Andrew Hudson Yang, Andrew Yang

CSE 158A Assignment 2

1. Data Exploration

The dataset that we chose to go with was the RentTheRunway dataset from the catalog of datasets found on the class website. The dataset originally has around 192544 data points and a format as shown below.


```
{'fit': 'fit',
 'user_id': '420272',
 'bust size': '34d',
 'item_id': '2260466',
 'weight': '137lbs',
 'rating': '10',
 'rented for': 'vacation',
 'review_text': "An adorable romper! Belt a
 'body type': 'hourglass',
 'review_summary': 'So many compliments!',
 'category': 'romper',
 'height': '5\' 8"',
 'size': 14,
 'age': '28',
 'review_date': 'April 20, 2016'}
```


As seen to the right, the data points are highly informative containing many different sources of information about the items and users that we can use to create a model. The abundance of potential features have motivated us to attempt a regression model that utilizes most of the information here to predict the rating. However, as this is a voluntary review made by people then we would certainly have to adjust the dataset to ensure that we do not have any invalid or missing values by either removing or adding default values to them. Thus, after choosing some features I wanted to focus on I removed any data points without values as shown in the code below, which lead to the final dataset we used to have 190971 data points.

```
data = [
    entry for entry in data
    if all(field in entry and entry[field] for field in ['rating', 'review_text', 'review_summary', 'size', 'age', 'review_date', 'rented for'])
]
```

Python

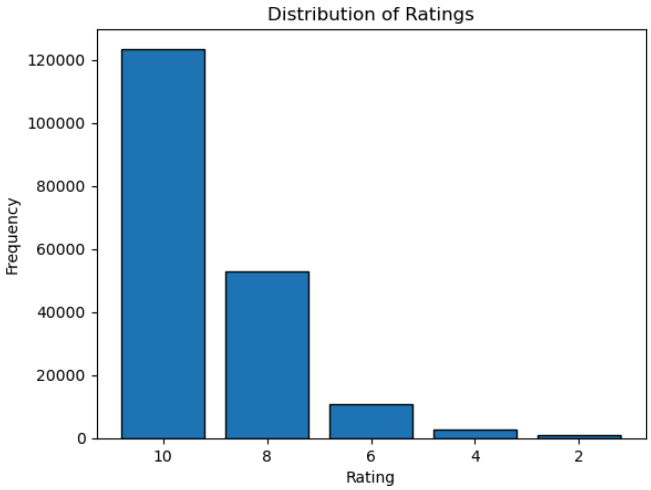
```
print(len(data))
```

Python

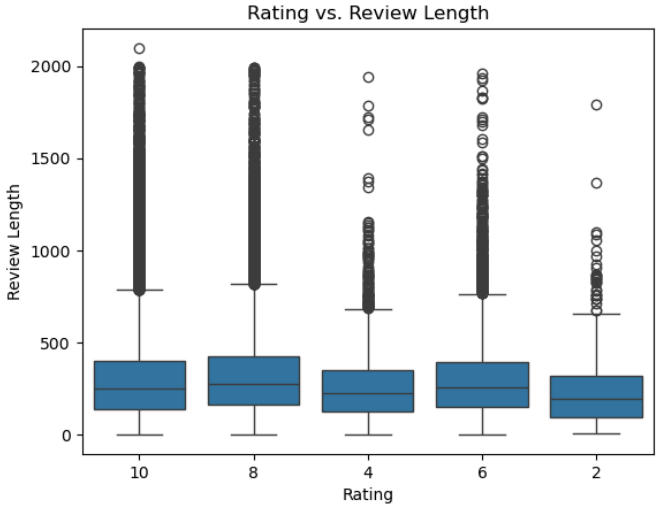
```
190971
```

The reason we chose to focus on these features particularly is because of how they can individually impact those who buy the product and reasons for why they buy it. For example, we can calculate the age and size interaction and observe how it influences the MSE of our model, next with the reviews we can observe many details about how the user thought about the item and use it for a sentiment analysis. For the “rented for” aspect, we can observe how people going

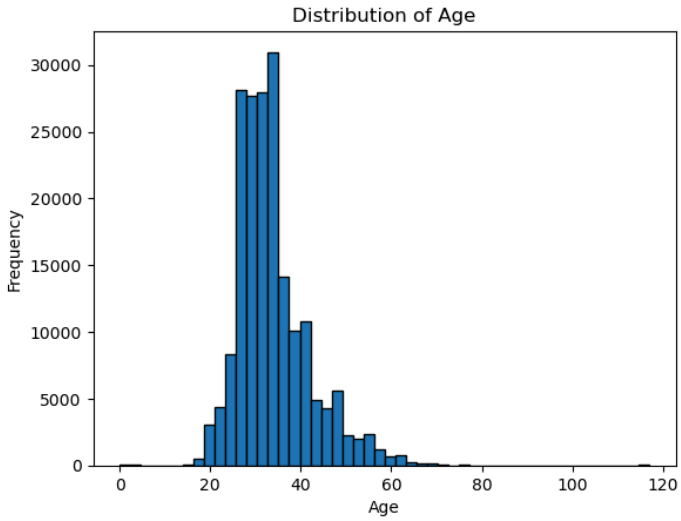
on vacation could potentially give higher reviews than people who buy clothes for a professional environment. Below we will attempt to analyze some aspects of our dataset to see



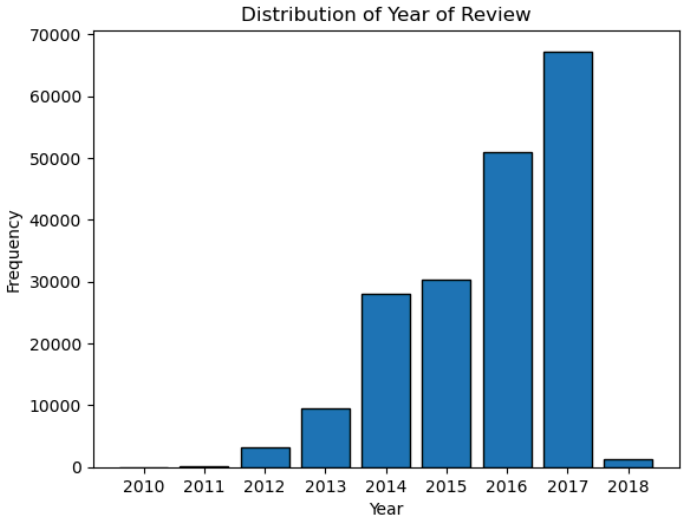
The chart above displays the distribution of ratings in our processed dataset. As we can see the ratings are skewed right with a median of 10 with over 120 000 reviews with that rating. This would likely imply that a model that always predicts the median would likely have a solid accuracy.



This boxplot shows an abundance of outliers that are way above the upper limit for each rating. The density of outliers above this range is seen to be most dense for rating 10 and gets progressively less dense as we go down in rating, which could imply that there is a higher chance a rating will be high if the review length is long or extremely long. Thus, we can test this in section 2.



This histogram of age shows that the median is around 30-40 year olds, so we could explore how the age distribution will affect our MSE in our regression models. The histogram also shows that there are also some older customers such as those over 60 years old.



The distribution of year of review here shows its skewed left and most of the reviews were done in 2017. Here we can potentially see how online reviews and online shopping in general became more popular or how the younger generation started to shop online more as the years went by. Thus, we can certainly attempt to see how the MSE of ratings would change using the year as a predictor.

2. Predictive Task

As mentioned in section 1, the predictive task we aim to study here is to predict the rating. The first model that we thought of using would be a feature driven regression model, which we will be testing using the MSE values we get from using a different set of features in each model. Some baseline models we used were ones that we did in class such as when we used the number of exclamation points to predict ratings. We also used some more basic ideas such as lengths of review_text and review_summary for more baseline models. For the training set and test set distribution we used a 80%/20% ratio. However, before looking at the MSEs of these models it is important to understand our data and the most important of which is the ratings.

```
unique_ratings = set(entry['rating'] for entry in data if 'rating' in entry)
unique_ratings
```

✓ 0.0s

```
{'10', '2', '4', '6', '8'}
```

The code above shows that the users actually only had a choice of rating the items at intervals of 2, so having an MSE of 2 is quite good as it would imply that we are predicting less than one level above or below the actual rating, since an MSE of 2 is the mean square error, so the actual error would be around 1.4

```
def feat1(d):
    return [1, len(d['review_text'])]
```

```
Xtrain = [feat1(d) for d in dataTrain]
ytrain = [int(d['rating']) for d in dataTrain]
Xtest = [feat1(d) for d in dataTest]
ytest = [int(d['rating']) for d in dataTest]
```

```
mod = linear_model.LinearRegression(fit_intercept=False)
mod.fit(Xtrain,ytrain)
predictions = mod.predict(Xtest)
mse = MSE(ytest, predictions)
mse
```

✓ 0.0s

```
2.0509410951628015
```

Above is our first baseline model, which utilizes only one feature which is the length of review_text and here we already have a decent MSE of 2.05, which is potentially only one level different from the true rating.

```
def feat3(d):
    return [1, len(d['review_text']), len(d['review_summary']), d['review_text'].count('!') + d['review_summary'].count('!')]
✓ 0.0s

Xtrain = [feat3(d) for d in dataTrain]
Xtest = [feat3(d) for d in dataTest]
mod = linear_model.LinearRegression(fit_intercept=False)
mod.fit(Xtrain,ytrain)
predictions = mod.predict(Xtest)
mse = MSE(ytest, predictions)
mse
✓ 0.2s
1.9290804311924041
```

Next we used the length of review_text, review_summary and counted the number of "!" in both similar to our first homework assignment, which serves as a good baseline model. This model already sees good improvement in our MSE at 1.93 compared to our first one. This suggests how observing characteristics reviews is a good direction to go in. However, before going into more detailed observations on reviews, we also have other features we can look at such as age, size, review dates and how they respond when combined.

To the right, the first MSE we see is when we added the age of the reviewer as a feature by taking its string variant and converting it into an integer. This feature showed very minimal changes in our MSE, compared to when we added the size of the item, which we formatted as an integer similar to age. This had a greater impact on MSE than just the age, which we could look at for later models.

```
def feat5(d):
    return [1, len(d['review_text']),
            len(d['review_summary']),
            d['review_text'].count('!') + d['review_summary'].count('!'),
            int(d['age']),
            ]
✓ 0.0s

Xtrain = [feat5(d) for d in dataTrain]
Xtest = [feat5(d) for d in dataTest]
mod = linear_model.LinearRegression(fit_intercept=False)
mod.fit(Xtrain,ytrain)
predictions = mod.predict(Xtest)
mse = MSE(ytest, predictions)
mse
✓ 0.3s
1.9287141659151328

def feat(d):
    return [1, len(d['review_text']),
            len(d['review_summary']),
            d['review_text'].count('!') + d['review_summary'].count('!'),
            int(d['age']),
            int(d['size'])
            ]
✓ 0.0s

Xtrain = [feat(d) for d in dataTrain]
Xtest = [feat(d) for d in dataTest]
mod = linear_model.LinearRegression(fit_intercept=False)
mod.fit(Xtrain,ytrain)
predictions = mod.predict(Xtest)
mse = MSE(ytest, predictions)
mse
✓ 0.3s
1.9252668614896955
```

```
    def feat(d):
        age = int(d['age'])
        size = int(d['size'])

        return [1, len(d['review_text']),
                len(d['review_summary']),
                d['review_text'].count('!') + d['review_summary'].count('!'),
                int(d['review_date'][-4:]),
                age,
                size
                ]
    Xtrain = [feat(d) for d in dataTrain]
    Xtest = [feat(d) for d in dataTest]
    mod = linear_model.LinearRegression(fit_intercept=False)
    mod.fit(Xtrain,ytrain)
    predictions = mod.predict(Xtest)
    mse = MSE(ytest, predictions)
    mse
 ✓ 0.3s

1.9142409555522135
```

Next we could observe the impact of the year that the review was made, which also had a significant impact on our MSE by around 0.01. This feature was motivated by the idea that the ratings given to certain clothing sales varied across the years due to potential events occurring during those times or how the popularity of online shopping in general increased and reached a wider and much larger audience .

```
def feat(d):
    age = int(d['age'])
    size = int(d['size'])
    age_size_interaction = age * size
    return [1, len(d['review_text']),
            len(d['review_summary']),
            d['review_text'].count('!') + d['review_summary'].count('!'),
            int(d['review_date'][-4:]),
            age_size_interaction
            ]
Xtrain = [feat(d) for d in dataTrain]
Xtest = [feat(d) for d in dataTest]
mod = linear_model.LinearRegression(fit_intercept=False)
mod.fit(Xtrain,ytrain)
predictions = mod.predict(Xtest)
mse = MSE(ytest, predictions)
mse
```

✓ 0.3s

```
1.9148641067333887
```

In this model we wanted to observe if age x size had a significant relationship with one another to create a significant effect on the MSE, but we did observe a slightly worse MSE when replacing age and size as features separately.

```
def feat(d):
    age = int(d['age'])
    size = int(d['size'])
    age_size_interaction = age * size
    sentiment_words = ['love', 'hate', 'perfect', 'amazing', 'awful', 'good', 'bad']
    sentiment_count = sum(d['review_text'].lower().count(word) + d['review_summary'].lower().count(word) for word in sentiment_words)
    return [1, len(d['review_text']),
            len(d['review_summary']),
            d['review_text'].count('!') + d['review_summary'].count('!'),
            int(d['review_date'][-4:]),
            age_size_interaction,
            sentiment_count
            ]
Xtrain = [feat(d) for d in dataTrain]
Xtest = [feat(d) for d in dataTest]
mod = linear_model.LinearRegression(fit_intercept=False)
mod.fit(Xtrain,ytrain)
predictions = mod.predict(Xtest)
mse = MSE(ytest, predictions)
mse
```

✓ 1.2s

```
1.838722954905515
```

Next we wanted to test how a potential sentiment analysis could work on this model by detecting generic review words such as love, hate, perfect and others to see how it impacts the MSE. This feature certainly had a significant impact as it lowered our MSE by around 0.08,

which led to us investigating more features related to the review_text and review_summary word choices for our final model in section 3.

```
rented_for_categories = ['date', 'party', 'formal affair', 'everyday', 'other', 'wedding', 'vacation', 'party: cocktail', 'work']

def feat(d):
    age = int(d['age'])
    size = int(d['size'])
    age_size_interaction = age * size
    sentiment_words = ['love', 'hate', 'perfect', 'amazing', 'awful', 'good', 'bad']
    sentiment_count = sum(d['review_text'].lower().count(word) + d['review_summary'].lower().count(word) for word in sentiment_words)
    unique_words = len(set(re.findall(r'\w+', d['review_text'] + " " + d['review_summary'])))
    rented_for_encoded = [1 if d.get('rented for', '').lower() == category else 0 for category in rented_for_categories]
    return [1, len(d['review_text']),
            len(d['review_summary']),
            d['review_text'].count('!') + d['review_summary'].count('!'),
            int(d['review_date'][-4:]),
            age_size_interaction,
            sentiment_count,
            ] + rented_for_encoded
Xtrain = [feat(d) for d in dataTrain]
Xtest = [feat(d) for d in dataTest]
mod = linear_model.LinearRegression(fit_intercept=False)
mod.fit(Xtrain,ytrain)
predictions = mod.predict(Xtest)
mse = MSE(ytest, predictions)
mse
```

✓ 4.1s

```
1.8279770862346172
```

This model is the last model where we tested the last feature to look at from the data set which was the “rented for” feature. After looking at all the unique values for it, which is shown in rented_for_categories, we created a one hot encoding for each entry. As seen by the MSE there is a reasonable impact from using this feature as it lowered the MSE by around 0.01.

With the analysis of features above, we can now move on to construct a few models for predicting ratings.

3. Models For Prediction

3.1 Featured Engineering Regression Model

In section 2 we observed how different features derived from the data can improve our MSE, but we concluded based on the ones that improved our MSE most were those involving the review_text and review_summary. Thus, to optimize our model as much as possible this motivated a more sentiment analysis approach to our feature engineering.

```
rented_for_categories = ['date', 'party', 'formal affair', 'everyday', 'other', 'wedding', 'vacation', 'party: cocktail', 'work']
def feat(d):
    age = int(d['age'])
    size = int(d['size'])
    age_size_interaction = age * size
    sentiment_words = ['love', 'hate', 'perfect', 'amazing', 'awful', 'good', 'bad']
    sentiment_count = sum(d['review_text'].lower().count(word) + d['review_summary'].lower().count(word) for word in sentiment_words)
    unique_words = len(set(re.findall(r'\w+', d['review_text'] + " " + d['review_summary'])))
    contains_recommend = 1 if 'recommend' in d['review_text'].lower() or 'recommend' in d['review_summary'].lower() else 0
    rented_for_encoded = [1 if d.get('rented for', '').lower() == category else 0 for category in rented_for_categories]
    positive_words = [
        "comfortable", "perfect", "stylish", "fashionable", "elegant", "cute", "fit",
        "love", "beautiful", "amazing", "flattering", "soft", "chic", "adorable",
        "gorgeous", "high-quality", "great", "favorite", "versatile", "recommended"
    ]
    negative_words = [
        "uncomfortable", "tight", "loose", "cheap", "poor", "bad", "ill-fitting",
        "hate", "dislike", "ugly", "stiff", "scratchy", "itchy", "heavy",
        "unflattering", "low-quality", "awkward", "disappointing", "not fit"
    ]
    positive_count = sum(d['review_text'].lower().count(word) + d['review_summary'].lower().count(word) for word in positive_words)
    negative_count = sum(d['review_text'].lower().count(word) + d['review_summary'].lower().count(word) for word in negative_words)

    return [1, len(d['review_text']),
            len(d['review_summary']),
            d['review_text'].count('!') + d['review_summary'].count('!'),
            int(d['review_date'][-4:]),
            age_size_interaction,
            sentiment_count,
            unique_words,
            contains_recommend,
            positive_count,
            negative_count
            ] + rented_for_encoded

Xtrain = [feat(d) for d in dataTrain]
Xtest = [feat(d) for d in dataTest]
mod = linear_model.LinearRegression(fit_intercept=False)
mod.fit(Xtrain,ytrain)
predictions = mod.predict(Xtest)
mse = MSE(ytest, predictions)
mse
✓ 8.5s
1.7520070352514556
```

Above is the final model used. Compared to the last model in section 2 we added several features relating to sentiment analysis to optimize our model. Firstly is the unique_words feature which counts the number of unique words found in both review_text and review_summary. Next is contains_recommend, which is a binary value checking if recommend was found in either the

text or summary because we believe recommend is a strong word that identifies a reviewer's thoughts on the item. Finally, the strongest feature in our testing was certainly the addition of positive_count and negative_count. This feature was reflective of words that we researched to be used widely in the clothing industry by customers and we differentiated them between positive and negative reinforcement words about the products. After the addition of these features we were able to make significant improvements to our MSE with a final MSE of 1.75 which is a 0.07 improvement from the last section 2 model. While there may be some overfitting due to the number of features, it should not significantly hinder the ability of this model as we did not use all the possible features in the data points and mainly focused on the review_text and review_summary features.

3.2 Latent Factor Model (Surprise)

We also tried creating a latent factor model (LFM) for this dataset due to how powerful it was in assignment 1. The motivation behind using this model was that we had a lot of data points, which would greatly benefit a LFM and that users who like certain items can greatly affect the ratings given to those items given a user.

```
uir_data = [entry for entry in data
            if all(field in entry and entry[field] for field in ['rating', 'user_id', 'item_id'])]
svd_data = [
    (d['user_id'], d['item_id'], int(d['rating']))
    for d in uir_data if 'user_id' in d and 'item_id' in d and 'rating' in d
]
```

We first had to create the appropriate data for our model, so we filtered out the data without users ids, item ids and ratings and formatted our data into svd_data so that we can use the surprise library to create our LFM.

Thus in order to best optimize our LFM parameters that give us the best MSE, we first created a training set, validation set and test set in a 60%/20%/20% split so that we can test the parameters. Then we ran a loop that would consider an array of different parameters for our model and would display the MSE for each of them on the validation set, while doing this it keeps track of all the results and returns the parameters with the MSE when running on the test set. The results can also be seen below the code. The following is the best parameter and MSE on the test set: {'n_factors': 1, 'n_epochs': 15, 'lr_all': 0.005, 'reg_all': 0.02} Test MSE with Best Parameters:

1.9177724617966976. As we can see the MSE is around 1.92 which is much higher than our regression model, so we considered this LFM unsuccessful after trying many different parameters for it.

```python
from sklearn.model_selection import train_test_split as sklearn_train_test_split
df = pd.DataFrame(svd_data, columns=['user_id', 'item_id', 'rating'])
train_df, temp_df = sklearn_train_test_split(df, test_size=0.4)
validation_df, test_df = sklearn_train_test_split(temp_df, test_size=0.5)
reader = Reader(rating_scale=(0, 10))
trainset = Dataset.load_from_df(train_df[['user_id', 'item_id', 'rating']], reader).build_full_trainset()
validation_data = [
    (row['user_id'], row['item_id'], row['rating']) for _, row in validation_df.iterrows()
]
test_data = [
    (row['user_id'], row['item_id'], row['rating']) for _, row in test_df.iterrows()
]
param_sets = [
    {'n_factors': 1, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 5, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 10, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 20, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 100, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},

    {'n_factors': 1, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 2, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 3, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 4, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 5, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},

    {'n_factors': 1, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 1, 'n_epochs': 30, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 1, 'n_epochs': 40, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 1, 'n_epochs': 50, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 1, 'n_epochs': 60, 'lr_all': 0.005, 'reg_all': 0.02},

    {'n_factors': 1, 'n_epochs': 1, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 1, 'n_epochs': 5, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 1, 'n_epochs': 10, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 1, 'n_epochs': 15, 'lr_all': 0.005, 'reg_all': 0.02},
    {'n_factors': 1, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02},
]
results = []
for params in param_sets:
    print(f"Testing parameters: {params}")
    model = SVD(**params)
    model.fit(trainset)
    predictions = model.test(validation_data)

    true_ratings = [pred.r_ui for pred in predictions]
    predicted_ratings = [pred.est for pred in predictions]
    mse = MSE(true_ratings, predicted_ratings)
    print(f"Validation MSE: {mse}")
    results.append({'params': params, 'mse': mse})

results = sorted(results, key=lambda x: x['mse'])
best_params = results[0]['params']
print("\nBest Parameters on Validation Set:")
print(best_params)

final_model = SVD(**best_params)
final_model.fit(trainset)

test_predictions = final_model.test(test_data)
true_ratings_test = [pred.r_ui for pred in test_predictions]
predicted_ratings_test = [pred.est for pred in test_predictions]
test_mse = MSE(true_ratings_test, predicted_ratings_test)
print(f"Test MSE with Best Parameters: {test_mse}")
```

Code for hyperparameter tuning our LFM.

```
Results from this code:
Testing parameters: {'n_factors': 1, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9383679124388509
Testing parameters: {'n_factors': 5, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9395895370302465
Testing parameters: {'n_factors': 10, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9418910307830286
Testing parameters: {'n_factors': 20, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9438374320605405
Testing parameters: {'n_factors': 100, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9537934896092721
Testing parameters: {'n_factors': 1, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9394530281058857
Testing parameters: {'n_factors': 2, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9393964578465621
Testing parameters: {'n_factors': 3, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9388948288975265
Testing parameters: {'n_factors': 4, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9402555228521374
Testing parameters: {'n_factors': 5, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9396448479383372
Testing parameters: {'n_factors': 1, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9384687005296064
Testing parameters: {'n_factors': 1, 'n_epochs': 30, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9527422647101267
Testing parameters: {'n_factors': 1, 'n_epochs': 40, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9670840136707632
Testing parameters: {'n_factors': 1, 'n_epochs': 50, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9914978323354304
Testing parameters: {'n_factors': 1, 'n_epochs': 60, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 2.0120083475639845
Testing parameters: {'n_factors': 1, 'n_epochs': 1, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9977512834513458
Testing parameters: {'n_factors': 1, 'n_epochs': 5, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.951487893378706
Testing parameters: {'n_factors': 1, 'n_epochs': 10, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9382445295575756
Testing parameters: {'n_factors': 1, 'n_epochs': 15, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9361078674189347
Testing parameters: {'n_factors': 1, 'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
Validation MSE: 1.9384840163842434
Best Parameters on Validation Set:
{'n_factors': 1, 'n_epochs': 15, 'lr_all': 0.005, 'reg_all': 0.02}
Test MSE with Best Parameters: 1.9177724617966976
```

Hyperparameter tuning logs for our LFM.

3.3 Strengths,Weaknesses, and Issues

The strengths of using the latent factor model was its ability to reduce all the features in the data points to just user-item interactions, which allows us to capture user preferences. It also would have been quite good for sparse data sets but in our case our dataset was quite dense with other features after data processing, which allowed it to favor a regression feature based model. The weaknesses of this model were that it had difficulty handling new items or users and potential overfitting. The overfitting issue was the reason why we chose to include a validation set for testing as when we tried to do so without one it caused extremely bad overfitting where we had MSEs as low as 0.1. Without the validation set, it seemed like our test set was also heavily overfitted, which we managed to fix by adding the validation set.

The strength of our regression model was based on dense features that allowed us to have highly informative data points to create better predictions. It also allowed us to identify which features were best at predicting the rating, which we concluded to be the review_summary and review_text. A big weakness could be overfitting, which we tried to remedy by focusing more on the review_summary and review_text instead of diving too deep into every possible feature. It is also a weakness not to be too familiar with the field of clothing that we are investigating as we could easily improve our sentiment analysis if we had a better understanding. We did not come across many significant issues on our regression model other than trying to figure out which features to focus on in order to avoid heavy overfitting. We attempted using several other features but we chose the ones that had the greatest impact in this assignment, as some features simply did not make any differences during our exploration.

4. Further Readings and Experimentation

In addition to the dataset we used, other related datasets have been studied in the context of explicit rating prediction, such as the MovieLens dataset and the Netflix Prize dataset. These datasets are widely used benchmarks for recommender systems research, offering denser interactions and larger user bases, making them more conducive to experiments with complex models. Unlike RentTheRunway, these datasets often represent entertainment consumption, where user preferences may exhibit different behavioral patterns.

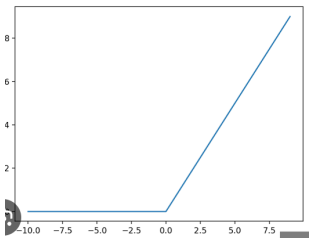
Approaches to explicit rating prediction typically fall into two broad categories: linear models, such as LFMs, and non-linear models, such as Neural Collaborative Filtering (NCF). Linear models assume that user-item interactions can be represented as a dot product in a latent space, but this linear assumption may fail to capture more complex interactions between users and items. Non-linear approaches, such as NCF, incorporate multi-layer perceptrons (MLPs) to capture complex and non-linear user-item interactions, thereby addressing the limitations of linear models.

The choice of method often depends on the characteristics of the dataset. Sparse datasets may favor simpler models due to the risk of overfitting in complex non-linear models, whereas denser datasets can fully exploit the representational capacity of neural architectures.

4.1: Neural Collaborative Filtering

One way to address the inflexibility of linearities is to introduce nonlinearities via multi-layer perceptrons.

To briefly touch over MLPs, at each layer, we have a set of weights (W, the weight matrix) that multiplies the input vector and adds bias (b). If all we do at each layer is multiplication and addition, then the entire calculation can be aggregated into one big matrix multiplication, which isn't what we want. Instead, after adding bias, we introduce an activation function (σ) that brings in non-linearity.



(ReLU, which is just a piecewise function $f(x) = x$ if $x \geq 0$ and 0 otherwise)

After applying the non-linearity, we pass the vector to the next layer. The shape of W at each layer depends on the dimension of the vector to be passed to the next level.

To apply MLP to the task of rating prediction, we can concatenate the latent factors of the users and items into a vector z to be used as input for our MLP. Then, z is passed through fully connected layers so at the last step, we can adjust the weight matrix w so that the output is simply one value, our expected rating:

$$h_1 = \sigma(W_1 z + b_1),\ h_2 = \sigma(W_2 h_1 + b_2),\ \dots,\ rating_{u,i} = \sigma(w^T h_n + b)$$

This introduces numerous parameters to learn, specifically the bias vectors and weight matrix at each layer. NCF uses MSE as its loss function, and though the specifics of calculating how to adjust parameters based on the loss is out of scope for this course, (the terminology is auto-differentiation for anyone interested, it is essentially a very long list of chain rules and derivatives) the general concept remains similar to many regression models in it tries to fit the model output as close to the training output.

The authors of NCF reported that their method demonstrated marginal but consistent improvements in recommendation task performance compared to traditional latent factor models. These improvements stem from the added flexibility provided by the introduction of non-linearities through multi-layer perceptrons, allowing for better modeling of complex user-item interactions.

Luckily existing libraries exist to help implement NCF. By introducing MLP, there is a crucial hyperparameter to tune other than the dimension of latent factors; namely, the structure of the layers of MLP. Based similarity that this dataset has with assignment 1's dataset, we utilized our previous experience and went with relatively low latent factors and neurons:

```
ncf = NCF(
    "rating",
    data_info,
    embed_size=2,
    n_epochs=5,
    lr=0.0001,
    lr_decay=False,
    reg=None,
    batch_size=128,
    use_bn=False,
    dropout_rate=None,
    hidden_units=(3, 3, 3),
    tf_sess_config=None,
)
```

The parameter hidden_units denote the number of nodes at each layer. The dimensions of both user and item is specified by embed_size, and all other hyperparameters are there as usual.

```
ncf.fit(
    train_data,
    neg_sampling=False,
    verbose=2,
    shuffle=True,
)
```

```
evaluate(
    model=ncf,
    data=test_data,
    neg_sampling=False,
    metrics=["rmse"],
)
```

The model is trained and tested on a 90/10 training/test split of the dataset. Hyperparameter tuning is done over the MLP dimensions (how many neurons at each layer), learning rate, epochs, and embedding size of users and items.

```
eval_pointwise: 100%|██████████| 2/2 [00:00<00:00, 592.71it/s]

{'rmse': 1.456076}
```

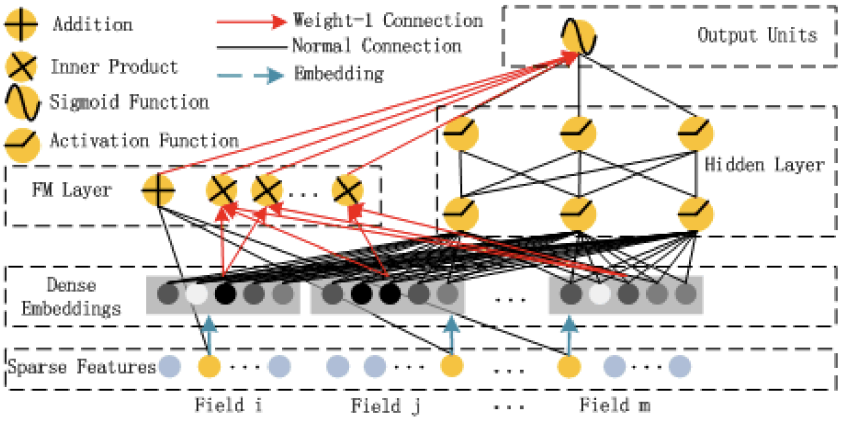
The best choice of hyperparameters gives us an RMSE of 1.456. Squaring that gets us an MSE value of around 2.12, which is worse compared to our LFM and linear regression models. Here, we first give a brief explanation of whether or not this result is in line with the results from the NCF paper, and save extensive conclusions for Section 5.

The relative sparsity of the dataset we used, compared to those used in the original NCF paper, imposes limitations on model complexity. Specifically, the embedding size and MLP

neuron count had to be constrained to prevent overfitting. As a result, there was no observed marginal improvement of NCF over LFM in our experiments, contrasting with the findings of the NCF authors. This discrepancy likely arises from differences in dataset characteristics, such as the higher density of user-item interactions in the datasets employed by the NCF authors, which could better leverage the expressive power of MLP structures.

4.2 Deep Factorization Machine (DeepFM)

DeepFM leverages a shared embedding layer to integrate feature representations from Factorization Machines (FM) and Deep Neural Networks (DNN). This architecture is designed to capture both low-order feature interactions (handled by FM) and high-order interactions (captured by DNN). By combining the strengths of both approaches, DeepFM overcomes the limitations of FM, which struggles with complex relationships, and DNN, which can require extensive feature engineering. This complementary design provides a more balanced representation of interactions within the data.



For an overview of the architecture of the model, we start at the shared embedding layer. Similar to NCF, input features are converted into dense vector embeddings $v_i \in R^k$, where k is the embedding dimension. These embeddings are shared/used by both the FM and DNN components. Next for the FM component, low-order feature interactions are captured through pairwise combination of embeddings:

$$FM\ Output = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle v_i, v_j \rangle x_i x_j.$$

The dot product $\langle v_i, v_j \rangle$ quantifies the interaction strength between features x_i and x_j, like in LFM. Concurrently, the DNN component captures high-order, non-linear interactions by passing concatenated embeddings through fully connected layers. In the end, the model predicts by summing the outputs of FM and DNN:

$$\hat{y} = FM\ Output + DNN\ Output$$

This architecture allows DeepFM to effectively integrate and represent both simple and complex feature interactions through its dual components.

To provide the model with features of potential significance, we included numerical columns such as size, weight, height, age, and bust size, alongside the user_id and item_id information. No additional feature engineering was performed, as we aimed to evaluate the model's ability to capture hidden relationships.

Hyperparameter tuning was done via grid search; in addition to common parameters such as batch_size, epochs, and learning_rate, we also tuned the embedding_dim, similar to what we did for NCF, which determines the dimensionality of feature embeddings. This is a critical parameter for DeepFM as it influences both the FM and DNN components of the model.

Through this grid search, we identified the optimal hyperparameters as follows:

```
Best Parameters: {'batch_size': 128, 'embedding_dim': 8, 'epochs': 1, 'learning_rate': 0.001}
Best Score: 1.9403688566379622
```

With an MSE of 1.940, the optimized DeepFM model performs worse than LFM but better than NCF. This result aligns with expectations, as DeepFM is conceptually positioned between LFM and NCF. The intermediate performance suggests that while DeepFM benefits from modeling linear relations, the dataset may lack sufficient or significant high-order interactions for neural network representations to provide a distinct advantage.

5. Results and Conclusion

In our final regression model we were able to achieve a final MSE of 1.75 by mainly using features derived from the review_test and review_summary, especially the sentiment analysis features. Other features such as the review year, rented_for and age x size interactions were used but they were not as significant as the sentiment analysis especially from positive and negative count. For our latent factor model comparison, we found that the best MSE we could achieve was 1.917 using the following parameters {'n_factors': 1, 'n_epochs': 15, 'lr_all': 0.005, 'reg_all': 0.02}. These parameters likely imply that our interaction data is quite simply with one factor driving most variation and is more sparse than dense. It also likely has a moderate complexity with no highly interlinked relationships between users and items. Our regression model likely worked better because of how dense our data points were in terms of the feature contents as we had filtered our data points to ensure this dense nature, thus feature engineering would inevitably drive good results, but it's not guaranteed for latent factor models to work just as well.

In contrast, the NCF model achieved a higher MSE of 2.12 using hyperparameters: {embed_size = 2, n_epochs = 5, lr = 0.0001, hidden_units = (3, 3, 3)}. This performance was significantly worse than both the regression and latent factor models, highlighting the limitations of NCF when applied to datasets with sparse user-item interactions.

Alternatively, the DeepFM model performed slightly better than NCF, with an MSE of 1.94 using the hyperparameters {'batch_size': 128, 'embedding_dim': 8, 'epochs': 1, 'learning_rate': 0.001}. While this was a small improvement over NCF, it still didn't surpass the latent factor model, suggesting that the high-order interactions, which DeepFM is designed to capture, were not strong enough in this dataset to significantly improve predictive performance. This could be attributed to the relatively sparse nature of the user-item interaction data, which didn't provide enough meaningful patterns for DeepFM to exploit.

We believe that the underperformance of both NCF and DeepFM is due to the sparse user-item interactions in the dataset. While DeepFM is designed to combine the benefits of Factorization Machines (FM) for low-order interactions and DNNs for high-order interactions, the dataset’s lack of sufficient high-order feature interactions limited the model's ability to outperform simpler models. Similarly, NCF, with its MLP layers, requires dense interaction data

to fully capture nonlinear relationships between users and items. In this case, the limited interaction data hindered the ability of both models to generalize effectively. By contrast, the regression model benefited from the dense set of engineered features derived from the review data, which were not as directly leveraged by the NCF or DeepFM architectures.

Complex models such as LFM, NCF, and DeepFM also fail to provide insightful knowledge based on their parameters, as they interact with embeddings of users and items that are projected to a high dimension. On the contrary, the simplicity of linear regression enables some information to be drawn based off the parameters, such as the relation between rating and the numerous features that we have engineered (sentiment score based off of exclamation marks and a dictionary).

To improve the performance of NCF and DeepFM on similarly sparse datasets, we can look to the suggestions of the authors of these models. First, incorporating additional explicit features, such as sentiment analysis and user features, directly into the model may help supplement the performance prediction. Second, pretraining the embedding layers with a simpler latent factor model could provide a more robust starting point for optimization. Finally, employing techniques such as transfer learning from larger, more robustly sampled datasets may also enhance the ability of these models to generalize in sparse environments.

Our results clearly demonstrate that the performance of recommendation models is highly dependent on the dataset.. Our regression model's performance shows that well-engineered features, particularly those derived from sentiment analysis, help boost performance the most. These features provided signals that captured user preferences, which allowed the model to generalize effectively. The latent factor model, though less effective than regression, revealed that the dataset primarily exhibited low-complexity interactions; the more complex models such as NCF and DeepFM, goes to show that a complex approach may not always be suitable for datasets of this nature.

To conclude, although advanced models like NCF and DeepFM hold promise for capturing complex patterns in dense datasets, simpler models with tailored feature engineering provide more reliable solutions for sparse datasets. Future works should explore augmenting sparse datasets with auxiliary features or leveraging pretraining and transfer learning techniques to enhance the generalizability of more complex approaches.