Question 1:

(a) Advantages:
→ Avoids huge MSE errors and potential numerical instabilities.
→ Faster training/optimization since we would have fewer training points.
→ Applicable to all training objectives/optimization routines trivially, as it is just a one-time pre-processing step.

Disadvantages:
→ Would skip over highly positive/negative training signals, i.e., when \( y < y_{\min} \) or \( y > y_{\max} \)
→ Hard to choose the range \([y_{\min}, y_{\max}]\) which could be dataset-specific.
→ For the test set, the model trained on the subset could generalize poorly if the test \( y \) lies largely outside \([y_{\min}, y_{\max}]\).

(b) Advantages:
→ Using a suitable transformation function would be less prone to numerical instabilities while optimizing the MSE.
→ Independent of the training objective, applies to all methods with just a single-pass over the dataset.

Disadvantages:
→ Hard to choose a suitable transformation function.
→ If the chosen transformation function is non-linear, it could hamper the performance of linear models like linear regression.
→ Could unnecessarily "squash" the remaining data, e.g., in a scenario where most of the ratings/y lie in \([1, 1.5]\) and some outliers are too low or high, a function like \( \log(y) \) would bring data points close to each other.
(c) Advantages:
→ Computed avoids the problem of large numbers.
→ Robust to even the largest outliers (indeo of value).
→ No tuning / hyper-parameters like scaling range, transformation function, etc. required.
→ Inexpensive transformation. Computed with just a single pass over the dataset.

Disadvantages:
→ Converting a regression problem to classification might unnecessarily change the scenario. E.g. in a movie-rating prediction task, converting it to binary classification can’t predict ratings (original task) on the test-set and predicting if rating > median doesn’t semantically imply much information.
→ For ranking scenarios: order \( x_i \) w.r.t their \( y_i \) values; the classification conversion loses out all intra-class ranking information, i.e. \( x_i, x_j \) with the same label (0 or 1), there is no possible ordering between them, unlike the original regression scenario where we can directly compare \( y_i \) and \( y_j \). Hence, a lot of information is unnecessarily lost.

(d) Advantages:
→ More robust to outliers than squared objective like MSE.
→ This method is independent of the dataset, and hence works trivially for all regression datasets.
→ No added hyper-parameter search like range-search \([y_{min}, y_{max}]\), transformation function, etc.
→ Zero added cost in terms of pre-processing.

Disadvantages:
→ The new objective function could possibly not have a closed-form solution, making optimization harder. E.g. MSE has closed-form solution and smoother gradients than MAE.
→ Not robust to large outliers. E.g. MAE could still suffer when \( y_i \rightarrow 10^6 \). Relatively better than MSE but worse than log-transformation or classification conversion, etc.
Question-3:

\[
\arg\min_{\theta_0} \frac{1}{N} \sum_{i=1}^{N} |\theta_0 - y_i| = L
\]

\[
\Rightarrow \arg\min_{\theta_0} \frac{1}{N} \left[ \sum_{i=1: \{y_i \leq \theta_0\}}^{R} \theta_0 - y_i + \sum_{j=1: \{y_j > \theta_0\}}^{N-R} y_j - \theta_0 \right]
\]

Differentiating the cost-function w.r.t \(\theta_0\) and equating to zero:

\[
\frac{\partial L}{\partial \theta_0} = 0 \Rightarrow \frac{1}{N} \left[ \sum_{i=1: \{y_i \leq \theta_0\}}^{R} 1 + \sum_{j=1: \{y_j > \theta_0\}}^{N-R} -1 \right] = 0
\]

\[
\therefore \frac{R}{N} = \frac{N-R}{N} \Rightarrow R = N/2
\]

Hence, there exists the same amount of \(y_i\)'s greater than \(\theta_0\) as there exist the amount of \(y_i\)'s lesser than \(\theta_0\).

\[
\Rightarrow \theta_0 \text{ is the median of } \{y_i\}_{i=1}^{N}.
\]

Question-4:

(a) I would try to collect features like:

- **Pickup location**: Passengers coming from a well-off / posh locality would tip more.
- **Drop location**: Passengers going to a posh locality (going back home) might tip more.
- **Trip Duration**: Long trips might lead to more generous tipping.
- **Passenger ID**: Knowing passenger's past history of tips might give a better estimate of the current tip amount.
- **Driver ID**: Driver's highly tipped before might have a better chance due to previous record.
- **Pickup time**: People might tip more during a certain part of the day from the other.
(b) - Location (Pickup/Drop): Instead of operating on latitude and longitude; if we are given a map of the city with demarcation of suburbs, etc.; I would use that. If not, I would cluster the latitude & longitude in my training data into some R-clusters based on Euclidian dist. Once I have the location’s corresponding suburb/cluster, I would represent it using a one-hot vector.

→ Trip Duration: A real number denoting the trip duration in hours. Then, I would normalize it by subtracting the mean duration and dividing by the std. deviation (from the train-set).

→ Driver/Passenger ID: Integers/Hashes.

→ Pickup time: There could be multiple things we want to capture while modeling time:
  1. Day of the year: Days like Christmas, Halloween, etc. are special & might have more tipping.
  2. Day of the month: People might tip more during start of month compared to month-end.
  3. Hour of the day: People might tip more during evenings compared to mornings.

[Note: I don’t want to go any further like minutes, seconds, etc. as I believe they won’t be that informative]

To model these diff. ideas, I would concatenate:
  1. One-hot repr. of day of year (365 dimensional)
  2. One-hot repr. of day of month (31 dim.)
  3. One-hot repr. of hour of day (24 dim.)

Hence, final rep.: \((365 + 31 + 24)\) dimensional.

(c) I’ll attempt the problem of tip prediction as a regression task with a MSE/MAE cost function.

To be specific, given a concatenated list of all the features discussed in (a) and (b), (let’s call it \(X \in \mathbb{R}^d\)); the optimization problem would be:
(let's call it $x \in \mathbb{R}^n$) so the optimization problem would be:

$$\arg\min_{x, \theta} \frac{1}{N} \sum_{i=1}^{N} \| y_i - (x + \bar{x} \cdot \bar{\theta}) \|_k^p + \lambda \| \theta \|_2^2$$

where "$k$" represents the $k$-norm:

$k=1$ for MAE and $k=2$ for MSE.

I am trying both depending on the number of outlier tips in the collected dataset.

It might be useful to transform the tips especially if there are some outliers in our data and we're using the MSE loss. Note that my model can handle fairly large tip-values because of the included global bias ($\lambda$) which has no regularizer.

→ We could throw away the points with too high tips if there aren't many. Or we could also simply replace it with a max value of a reasonably high tip-value. (Relative to the mean tip).

→ Transformation like $\log(x)$ might be unnecessary as they make the output non-linear and our tip-values would most likely not be extremely large.

**Question 5:**

This sort of classifier won't work as well as other classifiers like logistic regression because they optimize the squared difference between predicted and actual "y" whereas logistic regression uses a sigmoid function to get probabilities:

$$\arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \| x_i \cdot \bar{\theta} - y_i \|_2^2 \quad \text{vs.} \quad \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} s(y_{i=1}) \cdot \sigma(x_i \cdot \bar{\theta}) + s(y_{i=1}=0) \cdot (1 - \sigma(x_i \cdot \bar{\theta}))$$

Hence, the linear regressor would be much more prone to learning bad classifiers when data in one class has much less variance than the other. To be more concrete, see the plots below:
Hence, the linear regressor would be much more prone to learning bad classifiers when data in one class has much less variance than the other. To be more concrete, see the plots below:

Here, the red class has a much higher variance compared to the green class, making the blue linear regression line tilt more towards the red class (with no class imbalance). Hence, this resulted in 2 misclassifications. However, the brown logistic regression sigmoid curve is able to classify every point correctly.
Section 1 (Regression)

1.0.1 Helper functions

def parse(f):
    for l in gzip.open(f):
        yield eval(l)

def MSE(predictions, labels):
    differences = [(x-y)**2 for x, y in zip(predictions, labels)]
    return sum(differences) / len(differences)

def classification_metrics(pred, y):
    metrics = {}
    metrics["Accuracy"] = sum(pred == y) / len(y)

    TP = sum(np.logical_and(pred, y))
    FP = sum(np.logical_and(pred, np.logical_not(y)))
    TN = sum(np.logical_and(np.logical_not(pred), np.logical_not(y)))
    FN = sum(np.logical_and(np.logical_not(pred), y))

    metrics["TPR"] = TP / (TP + FN)
metrics["FPR"] = FP / (FP + TN)
metrics["TNR"] = TN / (TN + FP)
metrics["FNR"] = FN / (FN + TP)
metrics["BER"] = (metrics["FPR"] + metrics["FNR"]) / 2.0

for m in metrics: metrics[m] = round(100.0 * metrics[m], 2)

return metrics

def Jaccard(s1, s2):
    numer = len(s1.intersection(s2))
    denom = len(s1.union(s2))
    return numer / denom

[3]: dataset = list(parse("goodreads_reviews_comics_graphic.json.gz"))

print("# of data-points:", len(dataset))
print(dataset[1])

# of data-points: 542338
{'user_id': 'bafc2d50014200cda7cb2b6acd60cd73', 'book_id': '6315584',
'review_id': '72f1229aba5a88f9e72f0dcdc007dd22', 'rating': 4, 'review_text':
"I've never really liked Spider-Man. I am, however, a huge fan of the Dresden
Files. Jim Butcher is clever and sarcastic and probably the perfect choice to
pen a superhero novel. I really enjoyed this book!", 'date_added': 'Wed Aug 10
06:06:48 -0700 2016', 'date_updated': 'Fri Aug 12 08:49:54 -0700 2016',
'read_at': 'Fri Aug 12 08:49:54 -0700 2016', 'started_at': 'Wed Aug 10 00:00:00
-0700 2016', 'n_votes': 0, 'n_comments': 0}

[4]: def feature(d):
    dayFeat = [0]*7 # One hot encoding of day of week
    dayDict = {"Mon":0, "Tue":1, "Wed":2, "Thu":3, "Fri":4, "Sat":5, "Sun":6}
    dayFeat[dayDict[d["date_added"][3]]] = 1
    return [1, d["rating"], d["n_comments"]] + dayFeat[1:]

    X = [feature(d) for d in dataset]
    y = [len(d["review_text"]) for d in dataset]

[5]: model = sklearn.linear_model.LinearRegression()
model.fit(X,y)
yPred = model.predict(X)
mse = MSE(yPred, y)
print("MSE = ", mse)

MSE = 624989.9720071984

Question 1a / 2a  Observing the distribution of review lengths:
```python
[6]: p = sns.distplot(y, kde=False)
p.set_yscale("log")
p.set_xlabel("Review length")
p.set_ylabel("Frequency (log-scale)")
plt.show()
```

```python
[7]: plot_x, plot_y, points_kept = [], [], []
    for ymax in [10_000, 12_500, 15_000, 17_500, 20_000 ]:
        X_new, y_new = [], []
        for i in range(len(X)):
            if y[i] < 0 or y[i] > ymax: continue
            X_new.append(X[i])
            y_new.append(y[i])

        model = sklearn.linear_model.LinearRegression()
        model.fit(X_new, y_new)
        yPred = model.predict(X)
        mse = MSE(yPred, y)

        plot_x.append(ymax)
        plot_y.append(mse)
        points_kept.append(100.0 * (len(X_new) / len(X)))

plt.clf()
```
After first analyzing the distribution of review lengths in the dataset, I tried multiple reasonable values of \( ymax \) and validated the MSE on the entire, original dataset. This design choice of evaluating the MSE on the entire original dataset (and not \( y_{\text{new}} \)) is to be consistent across methods and see which strategies in Q1 perform the best.

As is evident from the plot, capping our dataset to only have smaller values of \( y \) doesn’t seem to be helping in terms of MSE on the entire dataset. The best MSE is when we have all the points (no deletion according to \( ymax \)).

**Question 1b/2b**

```python
plot_x, plot_y = [], []
to_try = [ 'log2', 'ln', 'log10' ]
for i, transformation in enumerate(to_try):
    transformation_fn, inv_transformation_fn = {
        'log2': (np.log2, lambda x: 2.0 ** x),
        'ln': (np.log, lambda x: np.e ** x),
```
'log10': (np.log10, lambda x: 10.0 ** x),
}

y_new = list(map(lambda x: transformation_fn(max(i, x)), y))

model = sklearn.linear_model.LinearRegression()
model.fit(X, y_new)
yPred = model.predict(X)
yPredConv = list(map(lambda x: inv_transformation_fn(x), yPred))

print("For transformation fn: {}, MSE on transformed/original set: {}/{}").

for transformation, MSE(yPred, y_new), MSE(yPredConv, y)
)

plot_x.append(i)
plot_y.append(MSE(yPredConv, y))

plt.clf()
plt.plot(plot_x, plot_y, color = "blue")
plt.xlabel("MSE on the entire dataset", color = "blue")
plt.xticks(plot_x, to_try)
plt.show()

For transformation fn: log2, MSE on transformed/original set:
4.026718799718/1.6267842017210838e+45
For transformation fn: ln, MSE on transformed/original set:
1.934434459551/1.626784201720888e+45
For transformation fn: log10, MSE on transformed/original set:
0.36485696617358454/1.6267842017221363e+45
I tried multiple transformation functions \((\log_2(x), \log_e(x), \log_{10}(x))\) and validated the MSE on (1) the transformed set, and (2) the entire, original dataset by inverting the predictions using the inverse transformation function \((2^x, e^x, 10^x)\).

Although the methods have a good MSE on the transformed set, as is evident from the plot, the transformation functions don’t generalize well to the full-dataset when the inverse transformations are applied.

**Question 1c/2c**

```python
median_y = np.median(y)
print("Median review length: ", median_y)
y_new = list(map(lambda x: int(x > median_y), y))

model = linear_model.LogisticRegression(C = 1.0, solver = 'lbfgs')
model.fit(X, y_new)
yPred = model.predict(X)
print("Classification metrics: ")
print(classification_metrics(yPred, y_new))
yPredConv = list(map(lambda x: median_y if x == 1 else 1, yPred))
print("MSE using simple conversion trick: ", MSE(yPredConv, y))
```

Median review length: 247.0
Classification metrics:
{'Accuracy': 52.31, 'TPR': 24.65, 'FPR': 20.16, 'TNR': 79.84, 'FNR': 75.35, 'BER': 47.75}
MSE using simple conversion trick: 835310.766164274

To evaluate the performance of this approach, I chose to: - Evaluate the classification performance on the transformed $y$. - Replace the predicted ones in $y_{\text{Pred}}$ to be the median review length and zeros in $y_{\text{Pred}}$ to be 1, and evaluate the MSE on the full dataset. The idea is simple: when the classification model predicts one, let’s assume the median length, and when it predicts 0, let’s assume the smallest length.

Both the classification and MSE performance are not good. The classification accuracy is almost random at 52% and the MSE performance is much worse than no transformation.

**Question 1d/2d**

```python
def train(x, y):
    x = np.array(list(map(lambda x: np.array(x), x)))
    y = np.array(y)
    track_mae = []
    track_mse = []

    theta_mae = np.zeros(x.shape[1])
    theta_mse = np.zeros(x.shape[1])
    y_mask = np.ones(x.shape[0])
    eta_mae = 10
    eta_mse = 0.1

    for epochs in tqdm(range(50)):
        # MAE
        preds_mae = x @ theta_mae
        y_mask[:] = 1
        y_mask[preds_mae > y] = -1
        gradient = np.mean(x * y_mask[:, None], axis = 0)
        theta_mae = theta_mae + (eta_mae * gradient)

        # MSE
        preds_mse = x @ theta_mse
        gradient = np.mean(x * (y - preds_mse)[:, None], axis = 0)
        theta_mse = theta_mse + (eta_mse * gradient)

        track_mae.append(MSE(preds_mae, y))
        track_mse.append(MSE(preds_mse, y))

    return track_mae, track_mse
```

```bash
track_mae, track_mse = train(X, y)
plt.clf()
plt.plot(list(range(len(track_mae))), track_mae, label = "trained using MAE")
plt.plot(list(range(len(track_mse))), track_mse, label = "trained using MSE")
plt.xlabel("Epochs")
```
I evaluate the performance of models (trained using MSE and MAE) on the full dataset. Note: I implemented training using MSE as well (in contrast to sklearn) to have a better comparison between the two models.

As we can see, the MAE model doesn’t perform well and gets a worse MSE than the model trained using MSE.

2 Section 2 (Classification)

Question 6
def feature(d):
    dayFeat = [0]*7  # One hot encoding of day of week
    dayDict = {"Mon":0, "Tue":1, "Wed":2, "Thu":3, "Fri":4, "Sat":5, "Sun":6}
    dayFeat[dayDict[d["date_added"]][3]] = 1
    return [1, int(d["rating"]), d["n_comments"], d["n_votes"]] + dayFeat[1:]

X = [ feature(d) for d in dataset ]
y = [ int(len(d["review_text"]) > 250) for d in dataset ]

model = sklearn.linear_model.LinearRegression()
model.fit(X, y)
yPredOrig = model.predict(X)
yPred = np.array(list(map(lambda x: int(x > 0.5), yPredOrig)))
print("Classification metrics when trained using linear regression:")
print(classification_metrics(np.array(yPred), y))

model = linear_model.LogisticRegression(C = 1.0, solver = 'lbfgs', class_weight='balanced')
model.fit(X, y)
yPred = model.predict(X)
print("Classification metrics when training using logistic regression:")
print(classification_metrics(yPred, y))

Classification metrics when trained using linear regression:
{'Accuracy': 54.71, 'TPR': 29.34, 'FPR': 20.38, 'TNR': 79.62, 'FNR': 70.66, 'BER': 45.52}

Classification metrics when training using logistic regression:
{'Accuracy': 55.62, 'TPR': 29.48, 'FPR': 18.73, 'TNR': 81.27, 'FNR': 70.52, 'BER': 44.62}

TASK: On the same Goodreads dataset, predict if the length of the review is greater than 250 given the rating, number of comments, number of votes, and the date that the review was added.

As we can observe, the logistic regression clearly outperforms the linear regression method on both Accuracy as well as the Balanced Error Rate (BER).

### 3 Section 3 (Recommender Systems)

# Utility data structures
reviewsPerUser = defaultdict(list)
reviewsPerItem = defaultdict(list)
usersPerItem = defaultdict(set)  # U_i from class slides
itemsPerUser = defaultdict(set)  # I_u from class slides
meanRatingPerItem = defaultdict(int)
for d in dataset:
    user, item = d['user_id'], d['book_id']
    reviewsPerUser[user].append(d)
    reviewsPerItem[item].append(d)
    usersPerItem[item].add(user)
    itemsPerUser[user].add(item)
    meanRatingPerItem[item] += int(d['rating'])

for item in meanRatingPerItem:
    meanRatingPerItem[item] = float(meanRatingPerItem[item]) / float(len(reviewsPerItem[item]))

ratingMean = sum([d['rating'] for d in dataset]) / len(dataset)
print("Mean rating: ", ratingMean)

Mean rating: 3.778138356523054

# This function should be re-defined for each of your model variants

for d in dataset:
    user, item = d['user_id'], d['book_id']
    reviewsPerUser[user].append(d)
    reviewsPerItem[item].append(d)
    usersPerItem[item].add(user)
    itemsPerUser[user].add(item)
    meanRatingPerItem[item] += int(d['rating'])

for item in meanRatingPerItem:
    meanRatingPerItem[item] = float(meanRatingPerItem[item]) / float(len(reviewsPerItem[item]))

ratingMean = sum([d['rating'] for d in dataset]) / len(dataset)
print("Mean rating: ", ratingMean)

Mean rating: 3.778138356523054

for d in dataset:
    user, item = d['user_id'], d['book_id']
    reviewsPerUser[user].append(d)
    reviewsPerItem[item].append(d)
    usersPerItem[item].add(user)
    itemsPerUser[user].add(item)
    meanRatingPerItem[item] += int(d['rating'])

for item in meanRatingPerItem:
    meanRatingPerItem[item] = float(meanRatingPerItem[item]) / float(len(reviewsPerItem[item]))

ratingMean = sum([d['rating'] for d in dataset]) / len(dataset)
print("Mean rating: ", ratingMean)

Mean rating: 3.778138356523054

# This function should be re-defined for each of your model variants

for d in dataset:
    user, item = d['user_id'], d['book_id']
    reviewsPerUser[user].append(d)
    reviewsPerItem[item].append(d)
    usersPerItem[item].add(user)
    itemsPerUser[user].add(item)
    meanRatingPerItem[item] += int(d['rating'])

for item in meanRatingPerItem:
    meanRatingPerItem[item] = float(meanRatingPerItem[item]) / float(len(reviewsPerItem[item]))

ratingMean = sum([d['rating'] for d in dataset]) / len(dataset)
print("Mean rating: ", ratingMean)

Mean rating: 3.778138356523054

# This function should be re-defined for each of your model variants

def predictRatingItem(user, item):
    ratings = []
    similarities = []
    for d in reviewsPerUser[user]:
        i2 = d['book_id']
        if i2 == item: continue
        ratings.append(d['rating'])
        similarities.append(Jaccard(usersPerItem[item], usersPerItem[i2]))
    if (sum(similarities) > 0):
        weightedRatings = [(x*y) for x, y in zip(ratings, similarities)]
        return sum(weightedRatings) / sum(similarities)
    else:
        # User hasn't rated any similar items
        return ratingMean

def predictRatingUser(user, item):
    ratings = []
    similarities = []
    for d in reviewsPerItem[item]:
        u2 = d['user_id']
        if u2 == user: continue
        ratings.append(d['rating'])
        similarities.append(Jaccard(itemsPerUser[user], itemsPerUser[u2]))
    if (sum(similarities) > 0):
        weightedRatings = [(x*y) for x, y in zip(ratings, similarities)]
        return sum(weightedRatings) / sum(similarities)
    else:
# Item hasn't been rated by similar users

```python
return ratingMean
```

```python
ts = random.sample(dataset, 1000)
sampleLabels = [d['rating'] for d in sample]
```

# Baseline prediction
```python
alwaysPredictMean = [ratingMean for d in sample]
baseline_mse = MSE(alwaysPredictMean, sampleLabels)
```

# Prediction using item-to-item similarity above
```python
itemCfPredictions = [predictRatingItem(d['user_id'], d['book_id']) for d in sample]
item_cf_mse = MSE(itemCfPredictions, sampleLabels)
```

# Prediction using user-to-user similarity above
```python
userCfPredictions = [predictRatingUser(d['user_id'], d['book_id']) for d in sample]
user_cf_mse = MSE(userCfPredictions, sampleLabels)
```

```python
print("MSE w/ baseline:", baseline_mse)
print("MSE w/ item2item similarity:", item_cf_mse)
print("MSE w/ user2user similarity:", user_cf_mse)
```

```
MSE w/ baseline: 1.2663718684468415
MSE w/ item2item similarity: 1.069613037462435
MSE w/ user2user similarity: 1.3132537537110962
```

Implemented the user-user similarity function rather than the item-item similarity function.

As we can see, the user-user similarity function gives a worse performance (in terms of MSE) compared to the item-item similarity function.

**Question 7 (b)**

```python
[16]: def predictRatingItemSubMean(user, item):
    ratings = []
similarities = []
    for d in reviewsPerUser[user]:
        i2 = d['book_id']
        if i2 == item: continue
        ratings.append(d['rating'] - meanRatingPerItem[i2])
        similarities.append(Jaccard(usersPerItem[item], usersPerItem[i2]))
    if (sum(similarities) > 0):
        weightedRatings = [(x+y) for x, y in zip(ratings, similarities)]
        return meanRatingPerItem[item] + (sum(weightedRatings) / sum(similarities))
    else:
```
# User hasn't rated any similar items

```python
return ratingMean
```

```python
sample = random.sample(dataset, 1000)
sampleLabels = [d['rating'] for d in sample]
```

```python
# Baseline prediction
alwaysPredictMean = [ratingMean for d in sample]
baseline_mse = MSE(alwaysPredictMean, sampleLabels)
```

```python
# Prediction using item-to-item similarity above
itemCfPredictions = [predictRatingItem(d['user_id'], d['book_id']) for d in sample]
item Cf mse = MSE(itemCfPredictions, sampleLabels)
```

```python
# Prediction using user-to-user similarity above
itemCfPredictionsSubMean = [predictRatingItemSubMean(d['user_id'], d['book_id']) for d in sample]
item_cf_mse_sub_mean = MSE(itemCfPredictionsSubMean, sampleLabels)
```

```python
print("MSE w/ baseline: ", baseline_mse)
print("MSE w/ item2item similarity: ", item Cf mse)
print("MSE w/ item2item similarity subtracted mean: ", item Cf mse_sub_mean)
```

MSE w/ baseline: 1.3589670876643785
MSE w/ item2item similarity: 1.0106793456340342
MSE w/ item2item similarity subtracted mean: 0.7901487136264821

Implemented the item-item similarity function where the mean rating of the item has been sub-
tracted, and I’m weighing the deviations.

As we can see, the item-item similarity function subtracted mean gives a much better per-
formance (in terms of MSE) compared to the original item-item similarity function.

**Question 7 (c)** Making the token vocabulary:

```python
[17]:
word_count = defaultdict(int)
punctuation = set(string.punctuation)

for d in dataset:
    for w in d['review_text'].split():
        w = ''.join([c for c in w.lower() if not c in punctuation])
        word_count[w] += 1

counts = [(word_count[w], w) for w in word_count]
counts.sort()
counts.reverse()
```
words = [x[1] for x in counts[50:1050]]
vocab_map = dict(zip(words, list(range(len(words)))))

bagOfWordsPerUser = {}
for d in tqdm(dataset):
    user, item = d['user_id'], d['book_id']
    if user not in bagOfWordsPerUser: bagOfWordsPerUser[user] = np.zeros(1000, dtype=np.float32)
    for w in d['review_text'].split():
        w = ''.join([c for c in w.lower() if not c in punctuation])
        if w in words: bagOfWordsPerUser[user][vocab_map[w]] += 1.0

HBox(children=(HTML(value=''), FloatProgress(value=0.0, max=542338.0), HTML(value='')))

[18]:
def predictRatingItemSubMeanMySimilarity(user, item):
    def similarity(set1, set2):
        set_1_vector = np.zeros(1000)
        set_2_vector = np.zeros(1000)
        for u in set1:
            set_1_vector += bagOfWordsPerUser[u] / float(len(reviewsPerUser[u]))
        for u in set2:
            set_2_vector += bagOfWordsPerUser[u] / float(len(reviewsPerUser[u]))
        set_1_vector /= float(len(set1))
        set_2_vector /= float(len(set2))
        return 1.0 - scipy.spatial.distance.cosine(set_1_vector, set_2_vector)

    ratings = []
similarities = []
    for d in reviewsPerUser[user]:
        i2 = d['book_id']
        if i2 == item: continue
        ratings.append(d['rating'] - meanRatingPerItem[i2])
        similarities.append(
            # Jaccard(usersPerItem[item],usersPerItem[i2])
            similarity(usersPerItem[item],usersPerItem[i2])
        )
    if (sum(similarities) > 0):
        weightedRatings = [(x*y) for x,y in zip(ratings,similarities)]
        return meanRatingPerItem[item] + (sum(weightedRatings) / sum(similarities))
    else:
        # User hasn't rated any similar items
return ratingMean

sample = random.sample(dataset, 1000)
sampleLabels = [d['rating'] for d in sample]

# Baseline prediction
alwaysPredictMean = [ratingMean for d in sample]
baseline_mse = MSE(alwaysPredictMean, sampleLabels)

# Prediction using item-to-item similarity above
itemCfPredictions = [predictRatingItem(d['user_id'], d['book_id']) for d in sample]
item_cf_mse = MSE(itemCfPredictions, sampleLabels)

# Prediction using user-to-user similarity above
itemCfPredictionsSubMean = [predictRatingItemSubMean(d['user_id'], d['book_id']) for d in sample]
item_cf_mse_sub_mean = MSE(itemCfPredictionsSubMean, sampleLabels)

# Prediction using user-to-user similarity above
itemCfPredictionsSubMeanMySimilarity = [predictRatingItemSubMeanMySimilarity(d['user_id'], d['book_id']) for d in tqdm(sample)]
item_cf_mse_sub_mean_my_sim = MSE(itemCfPredictionsSubMeanMySimilarity, sampleLabels)

print("MSE w/ baseline:", baseline_mse)
print("MSE w/ item2item similarity:", item_cf_mse)
print("MSE w/ item2item similarity subtracted mean:", item_cf_mse_sub_mean)
print("MSE w/ item2item similarity subtracted mean and my similarity:", item_cf_mse_sub_mean_my_sim)

In addition to the item-item similarity function where the mean rating of the item has been sub-
tracted (Question 7 (b)), here, I changed the similarity function to: - Instead of Jaccard similarity between the two sets of users, I computed the cosine similarity between the average bag-of-words features of the two sets:

\[
\text{similarity}(\text{set}_1, \text{set}_2) = \left( \frac{1}{|\text{set}_1|} \cdot \sum_{u \in \text{set}_1} f_u \right) \cdot \left( \frac{1}{|\text{set}_2|} \cdot \sum_{u \in \text{set}_2} f_u \right)
\]

where,

\[
f_u = \frac{1}{|\text{reviews}(u)|} \cdot \sum_{r \in \text{reviews}(u)} \text{bag-of-words}(r)
\]

For the design choices of computing the bag-of-words features: - Each word has been lower-cased and removed for punctuations. - I keep the 50-1050 words sorted according to count. I removed the top-50 words as stopwords.

As we can see, the item-item similarity function subtracted mean and my similarity gives a slightly better performance (in terms of MSE) compared to the original item-item similarity function subtracted mean. This indicates that even though with a token vocab. of 1000, the reviews are quite helpful to compute similarities amongst users.