1.

For this assignment, we are using the Steam dataset. While exploring the dataset, we noticed an abundance of data and identifiers that brought some ideas that correlated with previous implementations of this class. Throughout this dataset, we applied 3 datasets called Review Data v1(80000 entries), User and Item Data v1(25799 entries), and Item Metadata v2(32135 entries).

Review Data (image below) holds all the reviews specific to the user. Each iteration contains the user ID/URL, and each review contains the item_id, posted timestamp, helpfulness, yes/no recommendation, and review text.

User and Item Data (image below) holds the games owned by each user and descriptors. Each iteration has a username, item count, and steam ID, and each item has an ID, name, overall playtime, and playtime in the past 2 weeks.


```
{'user_id': 'MinxIsBetterThanPotatoes',
 'items_count': 371,
 'steam_id': '76561198004744620',
 'user_url': 'http://steamcommunity.com/id/MinxIsBetterThanPotatoes',
 'items': [{'item_id': '50',
   'item_name': 'Half-Life: Opposing Force',
   'playtime_forever': 256,
   'playtime_2weeks': 0},
  {'item_id': '240',
   'item_name': 'Counter-Strike: Source',
   'playtime_forever': 167,
   'playtime_2weeks': 0},
  {'item_id': '320',
   'item_name': 'Half-Life 2: Deathmatch',
   'playtime_forever': 3674,
   'playtime_2weeks': 0},
  {'item_id': '4000',
```


Through these 2 datasets, we found important properties such as user ID, item ID, review text, recommendation, and overall playtime. User ID and item ID are important because they create that user-item relation that we've seen throughout our coursework, and they provide context to the data within each user and item.

The last file provides additional features for games. This dataset holds the publisher, genres, app name, sentiment, title, url, release date, tags, review url, specs, price, early_access, id, and developer. We used most of them except time stamps.


```
{'publisher': 'Poolians.com',
 'genres': ['Casual', 'Free to Play', 'Indie', 'Simulation', 'Sports'],
 'app_name': 'Real Pool 3D - Poolians',
 'sentiment': 'Mostly Positive',
 'title': 'Real Pool 3D - Poolians',
 'url': 'http://store.steampowered.com/app/670290/Real_Pool_3D__Poolians/',
 'release_date': '2017-07-24',
 'tags': ['Free to Play',
  'Simulation',
  'Sports',
  'Casual',
  'Indie',
  'Multiplayer'],
 'reviews_url': 'http://steamcommunity.com/app/670290/reviews/?browsefilter=mostrecent&p=1',
 'specs': ['Single-player',
  'Multi-player',
  'Online Multi-Player',
  'In-App Purchases',
  'Stats'],
 'price': 'Free to Play',
 'early_access': False,
 'id': '670290',
 'developer': 'Poolians.com'}
```


The first two datasets were combined into a new data set called merged_data.


```
{'user_id': 'THEGHOST646',
 'item_id': '209160',
 'review_text': 'Simple review:Campaign
 'recommend': True,
 'playtime_forever': 351,
 'playtime_2weeks': 0}
```


This new merged_data set has 4200 entries, with 18000 unique user ids and 3000 thousand unique game ids. The distribution of Recommend/Does not Recommend across this dataset was the following.

```
Recommendation Distribution:
{False: 4753, True: 37457}

Recommend True Count: 37457
Recommend False Count: 4753
Percentage of Recommendations: 88.7396351575456 %
```

As you can see, the data is very imbalanced. For reasons described in part 2, we decided to balance the dataset where 50% of reviews are positive and the other 50% are negative.

```
New Recommendation Distribution:
{False: 4753, True: 4753}
```

```
Number of unique user IDs: 6886
Number of unique item IDs: 1780
```

2.

Predictive task: predict ‘recommend’ in (user, game,’ recommend’) dataset within 2 contexts: 1. recommender system 2. text & sentiment analysis.)

Baseline prediction of the ‘recommend’ field: Randomized classifier (a classifier that predicts randomly)

The first context for this prediction is the recommender system: given the data about game, or user-item interactions, without knowing anything about user review_text or hours played, would they recommend (like/dislike) this game? In this context, we are trying to recommend a game. For the second context, we used the same predictive task, of predicting the recommended field, within a different context: text metadata and sentiment analysis. In this context, we do have available data about what review text the user left for a game and how many hours they played that game. In this case, we are still predicting the same thing, but with different data and within sentiment analysis (for logistic regression, we analyze review length and hours played, for the n-gram we analyze the review_text, all to predict ‘recommend’ True/False).

First, we merged the data from the review file and the user and item data file based on matching the users from both. All of it was merged into a list that contained user_id, item_id, review text, recommendation status, and playtime data (image below).

```
balanced_merged_data[0]
```

```
{'user_id': '76561198046418559',
 'item_id': '316010',
 'review_text': "Good game overall. Finally can customize my deck how I like it
evious MTG games on steam. It's also free to play, with a fair f2p model. Will
 'recommend': True,
 'playtime_forever': 914,
 'playtime_2weeks': 0}
```

However, the merged data was imbalanced because 90% of the merged list was positive reviews that recommended the game. So if we made a predictor that always recommended a game, it would be 90% accurate. The accuracy of such a predictor would be misleading since F1 and precision values for the least distributed class (minority) would be very low. So, our main goal essentially was to keep good accuracy while improving the F1 score for the minority class.

Not only that, there are 42000 entries with these statistics: Unique user IDs: 18866 and Unique item IDs: 3054 which shows the sparseness of this data.

We grabbed the metadata for each item in item_metadata_set and took a subset of it such that item_metadata_set contains only metadata for games present in balanced_merged_dataset to reduce the computational cost. So from 30000 games in item_metadata, we now only use 1233 entries. The amount of entries in the item_metadata set is less than the overall number of games in the balanced_merged_data dataset for one reason: we merged by same game_name and game_id between two versions of the dataset, so that could result in some games from both subsets not appear (if for example a game_name in one dataset is present, but not in the other dataset.) We will fix this later on to shrink merged_data into its subset such that only games that are present in item_metadata appear in the subset. Our baseline on a balanced dataset is a dummy classifier which makes random predictions. The baseline is 50% accuracy (since the dataset is balanced 50/50 with 2 classes).

For our SVD model, we used user-item metrics to fit the model. We didn’t have to do any preprocessing for the data. All the model needed was users, items, and whether or not they recommended the items. This was done in the context of recommender systems.

For the logistic regression model with one-hot encodingsl (with some multi-class labels added later), we had a range of 3 to 5 features (with some of them isolated and tested individually.) To preprocess the data, we put the list with the data into pandas data frames to make them easier to work with. To assess the validity of the model, we have both an accuracy of correctness over all checks and a classification report that shows more in-depth the outcome of this model. We also have more information on the most "important features," which contain both tags and genres.

```
# Create a DataFrame
df = pd.DataFrame(combined_data)

# Quick price conversion
df['price'] = pd.to_numeric(df['price'].replace('Free to Play', 0), errors='coerce').fillna(

# Faster list handling
df['genres'] = df['genres'].apply(lambda x: x if isinstance(x, list) else [])
df['tags'] = df['tags'].apply(lambda x: x if isinstance(x, list) else [])
```

For the N-grams extraction model, we had to preprocess the review_text data. We converted everything to lowercase, removed punctuation, and removed stopwords that weren't helpful in the review. We generated the top 2,000 most frequent n-grams to use for the feature vector. The feature for this model was the counts of how many times each of the most frequent 2,000 n-grams appeared in each review. To assess the validity, we will do the same process which can be found in results part.

```
wordCount = defaultdict(int)
punctuation = set(string.punctuation)
stop_words = set(stopwords.words('english'))
for d in balanced_merged_data:
    review_text = d.get('review_text', '').lower()  # Default to empty string if key is missing

    # Remove punctuation
    r = ''.join([c for c in review_text if c not in punctuation])

    # Tokenize and filter stopwords
    ws = [w for w in r.split() if w not in stop_words]
    ws2 = [' '.join(x) for x in list(zip(ws[:-1],ws[1:]))]
    ws3 = [' '.join(x) for x in list(zip(ws[:-2],ws[1:-1],ws[2:]))]
    ws4 = [' '.join(x) for x in list(zip(ws[:-3],ws[1:-2],ws[2:-1],ws[3:]))]
    ws5 = [' '.join(x) for x in list(zip(ws[:-4],ws[1:-3],ws[2:-2],ws[3:-1],ws[4:]))]
    for w in ws + ws2 + ws3 + ws4 + ws5:
        wordCount[w] += 1

counts = [(wordCount[w], w) for w in wordCount]
counts.sort()
counts.reverse()
```

3.

For both contexts, we used Logistic Regression with various features (one hot encodings, multi-class categorical variables, n-grams.) Specifically for the recommendation prediction task, we also used matrix factorization and cosine similarity with various ranking methods for predicting whether a user would recommend (like/dislike) a game.

1) Randomized classifier (50% on all metrics in [F1, Recall, Accuracy] as well as BER)
2) Logistic Regression on review text length + 2 * hours played (testing predicting task with text length and hours played, since this data would not be available when guessing if we should recommend a game to a user(not a recommender task)
3) SVD based on user-item matrix (is recommender task), matrix factorization
4) Cosine Similarity, collaborative filtering (is recommender task)
5) Logistic Regression with different futures (is recommender task)
    a) Game Genres + game tags as one hot encoding
    b) Game Genres + game tags + metascore + publisher + developer as one hot encoding & category labels
    c) Genres + tags + metascore + publisher + developer + sentiment as one hot encoding & category labels
6) N-gram + Logistic Regression: prediction of 'recommend field' (not a recommender task in that we can't build a recommender system with it, but is still a model that predicts whether a user will recommend a game)

Recommendation context:
SVD, despite having slightly worse accuracy than some Logistic Regression models, was the one that required the least amount of data (just user-item interactions) and no complicated feature vectors. Cosine, compared to SVD, performed very badly on test data due to the dataset being extremely sparse (each user recommended only a few games on average, as well as many games not having more than 1 review.) The Logistic regression predictors performed just slightly better than SVD and provided a slight boost in overall accuracy as well as recall, and F1 scores for predicting whether a user would dislike the game. Logistic Regression for this context was optimized by isolating and adding relevant features into feature vectors. The main advantage of these models is that they can be used to actually recommend a game to a user if the prediction of recommendation is true. We optimized 4 parameters for the model: n_factors=100, n_epochs=20, lr-all=0.005, and reg_all=0.02.

Sentiment Analysis context:
The models predicted the same value 'recommend': True/False. The first was very simple logistic regression with review length and hours_played as a feature and the second used n-grams in review_text as features. The first one was optimized by adding the review_text instead of relying on the text length. The advantage of these models is that they provide better accuracy in the prediction of whether a user clicked 'recommend/do not recommend', which can be used for training chatbots or sentiment analysis models, but it cannot be used for recommending games to users since we already have the review text and hours played explaining that user already purchased a game and recommending at this point is pointless.

The last variation of our model, despite being quite complicated in terms of feature vectors, proved to be very valuable compared to the baseline. For example, if certain genres combined with certain tags correlate highly with users who recommend games, the model will predict that the new user will like the game. These features were combined into a logistic regression. For scalability purposes, we had to cap the one-hot encodings to a max length of 1000 to reduce dimensionality and prevent things like overfitting. For example, if we included every unique tag or genre in the dataset, it would become sparse and cause overfitting to insignificant tags and genres. Again, we optimized the regularization parameter finding C=1 had the best results. The careful analysis of the features showed that their combination showed slightly better results as a feature vector comparing to each of them being the only feature in the feature vector. When it comes to multi-class variables like sentiment, it takes values of "negative", "mixed", "very positive", etc. We gradually added more complexity to our model. With these 3 different levels of one hot encoding with additional features and labels for multi-class variables, we were able to make stronger connections and predictions which achieved a greater percent accuracy. Overall this model was strong enough, given the 50/50 balance and the sparse data that was created.

Given the nature of the data and that it is very sparse (for example, most of the users have just a few ratings, and the rest of the matrix is just hundreds of 0), that implied that binary recommendation prediction with jaccard/cosine similarity was challenging.

We could not go above 70% accuracy despite many attempts to optimize hyperparameters and try different classifiers (like NB). So we decided to experiment with sentiment analysis for the

same predictive task (whether a user likes or dislikes the game) but in different context . The last model we ran was another logistic regression, but instead, the features were based on n-grams. The idea was that certain n-grams would appear more in positive reviews and so if you count the number of the most common engrams you encounter in each review, this can correlate to a user either recommending or not recommending a game. We optimized using the regularization parameter and found that C=1 produced the highest percent accuracy. We processed up to 5-grams for the text, which was quickly ballooning up memory. So for scalability reasons, we limited just to checking for the top 2,000 most common n-grams.

4.

Self-attentive sequential recommendation
Wang-Cheng Kang, Julian McAuley
ICDM, 2018

Generating and personalizing bundle recommendations on Steam
Apurva Pathak, Kshitiz Gupta, Julian McAuley
SIGIR, 2017

The steam dataset was used by the professor and his colleagues in the above articles, in "Self-attentive sequential recommendation" they used a neural network with self-attention, this was compared against baselines which included: popularity based ranking, Markov chains, RNNs, CNNs, and Bayesian Personalized Ranking. The neural network with self-attention ended up with the highest score out of the other methods. The metric used was the hit rate for top ten and NDCG for top ten which assigns larger weights to higher positions. The dataset was also used to test personalized bundling for steam games using BPR (Bayesian Personalized Ranking).

The paper "Text Classification Algorithms: A Survey" by Kowsari, K., Meimandi, K. J., Heidarysafa, M., Mendu, S., Barnes, L. E., & Brown, D. E. discusses text classification approaches as well as preprocessing methods. The paper describes many methods for preprocessing, including tokenization, which we used in the n-grams model, basic cleaning like removing punctuation and stemming, which we didn't end up using. It also discusses the n-gram technique in particular, talking about how, most commonly, the 2-gram and 3-gram features are used because they offer a better understanding compared to 1-gram.

It is a bit difficult to gauge the difference in our model compared to the others described earlier due to the fact that they all use different metrics. Our metrics were basically just measuring the accuracy and F1 of different models, while in the papers, they measure things like hit rate for top ten.

5.

To reiterate, we started with the baseline model and a simple logistic regression to answer the question of whether hours in the game and review length can predict whether a user likes or dislikes a game (it did not). Then we gradually added more features for our logistic regression as well as tried other approaches like SVD and Cosine to one hot encoding and finished our data analysis with an n-grams model. Again, we want to emphasize that n-gram and the very first logistic regression model (number 1) were not used in the recommendation context since they relied on the review of the product itself rather than information about a product. That is why we had two contexts for the same prediction task: predicting 'recommend' in an arbitrary (user,pair,'recommend') entry. The baseline model was perfect compared to because it was created to fit an exact randomized baseline (all scores, accuracy, and BAR were 0.50 which is exactly 50%.) For the very first logistic regression, our features had review text length and overall playtime. This small increase in accuracy demonstrated that the length of review text and playtime held no statistical significance since BER was equal to 0.49.

```
Logistic Regression Predictions:
              precision    recall  f1-score   support

       False       0.60      0.47      0.53      1209
        True       0.55      0.68      0.61      1168

    accuracy                           0.57      2377
   macro avg       0.58      0.58      0.57      2377
weighted avg       0.58      0.57      0.57      2377
```

One thing to note about the 50-50 split for the dataset is that it results in lower accuracy but better precision, recall, and f1-score compared to the unbalanced dataset. Intuitively you can imagine that if a dataset is 90% positive and 10% negative, then predicting always true would give you 90% accuracy however we neglect the negative class, and when calculating the precision, recall, and f1-score for the negative class, we will get very bad F1 score for all of them. This demonstrates that when evaluating a model, you have to use different metrics that can capture the whole picture, and balancing the dataset is paramount.

One method that failed was our cosine model. On the whole dataset, the prediction accuracy was 95%, but despite numerous approaches to do ranking (Eq 4.20, 4.21, 4.22 from textbook) as well as using hybrid approaches for user-item similarity with a balanced dataset, the sparsity of data resulted in 10-50% accuracy across various train data sets sizes. So we decided to leave off the Cosine approach.

Given the logic that we won't have the review text and hours played when recommending a game to a user, we drop those and try to make a predictor based on the latent factor model, specifically SVD. We make a matrix of all user-item interactions with that Reader format

```
surprise_dataset = Dataset.load_from_df(df[['user', 'item', 'rating']], reader)
```

```
# Split the dataset
trainset, testset = train_test_split(surprise_dataset, test_size=0.2)

# Use SVD algorithm
model = SVD(
    n_factors=100,  # Reduce dimensionality
    n_epochs=20,    # Limit training iterations
    lr_all=0.005,   # Learning rate
    reg_all=0.02    # Regularization to prevent overfitting
)
```

With these results:

```
Classification Report:
              precision    recall  f1-score   support

         0.0       0.65      0.72      0.68       918
         1.0       0.71      0.64      0.67       984

    accuracy                           0.68      1902
   macro avg       0.68      0.68      0.68      1902
weighted avg       0.68      0.68      0.68      1902
```

Then, we did one-hot encodings models.
a)We built a feature matrix using MultiLabelBinalizier to make sparse one hot encodings for genres and tags and used this split + logistic regression:

```
# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)

# Train with fewer iterations and stronger regularization
model = LogisticRegression(max_iter=1200, C=1)
model.fit(X_train, y_train)
```

b) Next, we did exact same split but now also added developer and publisher information using the same one-hot encoding to the previous future matrix as well as metascore. For games that did not have Metascore we used global average Metascore across all games in the training dataset.

```
--- Logistic Regression ---
Accuracy: 0.6937002172338885

Classification Report:
              precision    recall  f1-score   support

           0       0.70      0.64      0.67       662
           1       0.69      0.74      0.72       719

    accuracy                           0.69      1381
   macro avg       0.69      0.69      0.69      1381
weighted avg       0.69      0.69      0.69      1381
```

With Metascore being a very important feature. We run the same logistic regression model using metascore alone, given its high importance, but that pushed the accuracy back to 65-67% range, implying some importance of developer and publisher.

Next, we also decided to add the sentiment part of the data, which takes values of “Negative”, “Mixed”, “Very Positive”, etc. This approach would not work if we tried to recommend a brand new game with 0 reviews, but in general, for most cases where games already have some reviews from early release, it still makes sense. This time we used label encoding for developer, publisher, and sentiment to treat them as multi-class categorical variables. The Logistic Regression with 2000 iterations and C = 1.3 yielded the best results on average compared to other Logistic Regression hyperparameter combinations as well as other classifiers. The resulting accuracy was roughly 70 percent using same test and train splits.

```
--- Logistic Regression ---
Accuracy: 0.7031136857349747

Classification Report:
              precision    recall  f1-score   support

           0       0.69      0.64      0.66       637
           1       0.71      0.76      0.73       744

    accuracy                           0.70      1381
   macro avg       0.70      0.70      0.70      1381
weighted avg       0.70      0.70      0.70      1381
```

As you can see, the precision and recall for the 1-class (likes) went up by a few decimals, while staying nearly the same for the 0-class (dislikes).

Finally, looking at our n-grams model, it was the best predictor model with an accuracy of 76%. Compared to our other models, this seems to have the most success indicating that the review text itself was the most significant data in our prediction. When it comes to gaming, many positive key text indicators show whether someone enjoys and recommends a game or not. So by applying the text through our model, we were able to see that through our recommendation accuracy.

```
]: mod = LogisticRegression(C=1)
   mod.fit(X_train, y_train)

   predictions = mod.predict(X_test) # Binary vector of predictions
   correct = predictions == y_test
   print(sum(correct) / len(correct))
```

```
0.7660917122423223
```