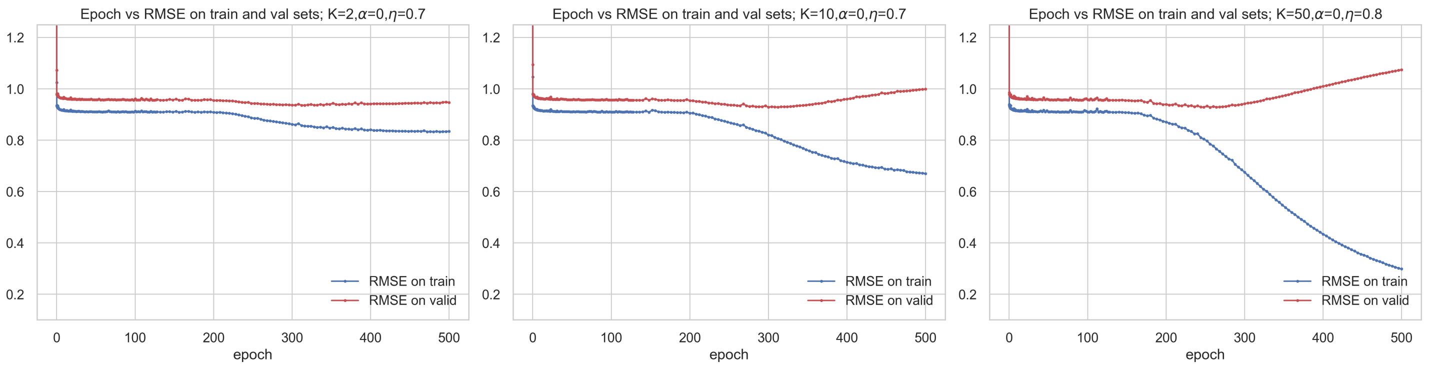
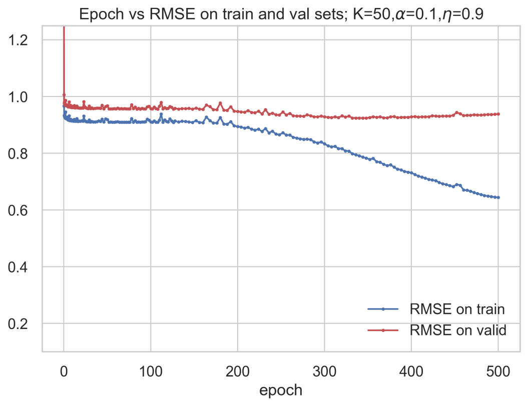
Problem 1

1a



As K gets bigger, it seems that overfitting increases for higher epochs. Above 250 or so epochs, the larger the K, the larger the overfit (difference between training and validation set performance). At $K = 2$, the "best" validation set performance is anywhere from 250 to 500 epochs. At $K = 10$, this range shifts to 250 to 325 epochs. At $K = 50$, this range is limited much more, to somewhere around 250 epochs. In terms of the performance within these ranges, it doesn't change much with different K values, always hovering around 0.93 RMSE. For step size, we chose sizes of 0.7 for K of 2 and 10, and 0.8 for K of 50. We chose our η by starting with a high step size (which initially converged), and bringing down our step size until we stopped diverging. At that point, we had the graphs we have now, which show obvious signs of overfitting, so we stopped.

1b



To penalize the model, we chose $\alpha = 0.1$ out of the range $[0.001, 0.01, 0.05, 0.1, 0.5, 1]$. We wanted to choose a larger α in order to dissuade overfitting in our model, but other αs higher than 0.01 did not show improved performance, showing that we may have been over-penalizing. We chose a step size of 0.9. While a higher α value does not inherently decrease step size, it does dampen weights and can effectively lead to longer training time. To account for this, we increased our learning rate a bit from our results in part a until we got a result that clearly showed overfitting. While the best validation set performance is not all that different from the $\{K = 50, \alpha = 0, \eta = 0.9\}$ model, validation set performance is significantly better in this model for higher epochs. In the $\{K = 50, \alpha = 0, \eta = 0.9\}$ model, validation and train error diverge dramatically (validation error increases, train error decreases) after about 250 epochs, whereas with an α of 0.1, validation error stays low until 500 epochs. However, we still opted for the weights achieved after epoch 344, as this was the minimal validation RMSE achieved. This is also a good epoch to stop at, since training error diverges dramatically past that point.

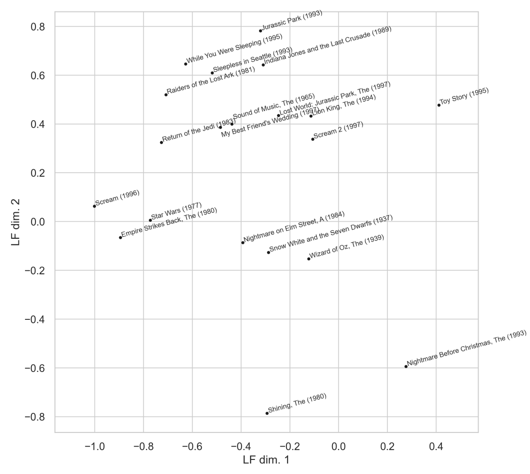
1c

model	RMSE train	RMSE test	RMSE val	MAE train	MAE test	MAE val
K=50, $\alpha = 0.100, \eta = 0.900$	0.786	0.927	0.924	0.618	0.720	0.726
K=50, $\alpha = 0.000, \eta = 0.800$	0.785	0.930	0.928	0.616	0.723	0.729
K=10, $\alpha = 0.000, \eta = 0.700$	0.857	0.940	0.934	0.676	0.732	0.737
K=2, $\alpha = 0.000, \eta = 0.700$	0.855	0.943	0.936	0.673	0.732	0.736

Note that all models were trained with 500 epochs. Focusing on RMSE, we recommend 50 factors ($K = 50$), as they both have the lowest validation and test RMSE values. That is, they perform best on heldout data, so we can be more confident that the model will accurately learn correlations between movies. If we look at test and validation MAE instead, our preference does not change as the ranking is the same for that metric.

1d

In general, we observe that movies with similar genres or themes tend to have similar learned embeddings. For instance, romantic comedies such as *Sleepless in Seattle* and *While You Were Sleeping* cluster closely, as expected. Similarly, the *Jurassic Park* series shows tight grouping, reflecting their shared genre and popularity. This is good, as a competent recomender would be expected to group movies of the same genre together. However, some genres, like family movies (*Toy Story*, *The Wizard of Oz*, *The Nightmare Before Christmas*), are more scattered, suggesting that our $K = 2$ model struggles to fully capture genre similarity in some cases.We also notice that very popular movies, which have many ratings, tend to be embedded closer together. Because our collaborative filter model is based only on user behavior, not on movie content, it's natural that some different-genre movies end up close together if they are similarly popular or frequently co-rated. Conversely, relatively less popular movies, such as *A Nightmare on Elm Street*, are less consistently placed. That is, its embedding is not as close those of other horror films like *Scream.* Overall, the embeddings reflect genre and popularity trends reasonably well, but there are some inconsistencies, likely due to data sparsity.



Problem 2

2a

Our proposed method uses the SVD++ matrix factorization algorithm from the Surprise Python library. We first split our known data into an 80/20 training/testing split, setting aside data so we can assess heldout performance later on. We then used `surprise.RandomizedSearchCV` to perform hyperparameter tuning over four different hyperparameters, using 5-fold cross-validation.

SVD is similar to the LF model from Problem 1, in that it can create embeddings that accurately create ratings for particular movies for particular users, given sparse matrices. Although the training objective is similar to the LF model from Problem 1, we can learn better estimates for the bias per item and user, as these are updated via SGD just like the matrix entries. So, we would be able to recreate the matrix with higher accuracy with these learned biases. We found from a paper by Yehuda Koren[1] that SVD++ improves on SVD via introducing the implicit ratings for each user to the model. With this, we can improve performance of recommendation for users with sparse ratings. This is because SVD introduces a term per user that indicates what movies the particular user has rated, and averages that information to learn the signal, not the noise, from those sparse ratings.

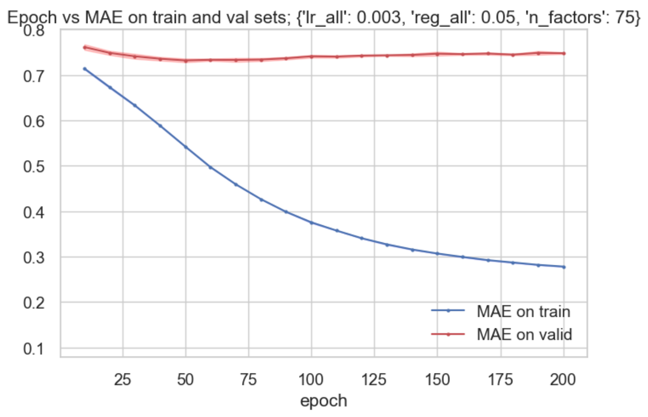
Hyperparameter	**Values**
`n_factors`	$[50.000, 75.000, 100.000]$
`lr_all`	$[0.003, 0.007, 0.010, 0.100]$
`reg_all`	$[0.050, 0.100, 0.200, 0.300]$
`n_epochs`	$[50.000, 100.000]$

We wanted to train the SVD++ model with a large range of hyperparameters to ensure model performance was prone to neither overfitting nor underfitting. We defined the range as such, prioritizing larger amounts of dimensions than in problem 1, as we saw that higher dimensions outperformed the lower ones (i.e. $K = 50$ outperformed $K \in \{2, 10\}$). Because of our emphasis in lower dimensions, we wanted to apply higher strength penalties, so opted for those around the magnitude 10^{-1}. Since higher strength penalties cause weights to be smaller, we also opted for faster learning rates. We selected the amount of epochs to be lesser than that chosen in problem 1, as we saw that LF models from Problem 1 all benefited from early stopping, implying that `n_epochs` could be safely set lower.

Training an SVD++ model is typically more computationally intensive, however. We mitigated this by using RandomizedSearchCV, which samples random combinations of hyperparameters instead of exhaustively testing all configurations (as in grid search), significantly reducing training time. We used 5 folds and 20 iterations of parameter searches. After selecting the best model, we retrained it on the full training data to maximize performance and avoid underfitting. Later, we used the heldout test set to evaluate the downstream metric, mean absolute error (MA).

[1] Y. Koren, R. Bell, and C. Volinsky, *Matrix factorization techniques for recommender systems, Computer, vol. 42, no. 8, pp. 30–37, 2009. https://people.engr.tamu.edu/huangrh/Spring16/papers_course/matrix_factorization.pdf*

2b



As shown in the figure, we found that the MAE on the test set stopped improving after 50 epochs, despite the MAE on the training set continuing to go down, signaling overfitting. Therefore, we opted to stop training early, at 50 epochs, to decrease risk of overfitting.

2c

Model	Test MAE (Dev Set)	Leaderboard MAE
Problem 2 Method	0.7311	0.7198
Problem 1 Method ($K = 50$)	0.7200	–
SVD	0.7488	–
SVD++ with augmented data	0.7343	–
KNNBaseline	0.7361	–

Note that the Problem 2 method is `surprise.SVDpp` with parameters `lr_all` = 0.003, `reg_all` = 0.05, `n_factors` = 75, `n_epochs` = 50. The Problem 1 method is `CollabFilterOneVectorPerItem` with parameters `n_factors` = 50, $\alpha = 0.100$, $\eta = 0.900$, and `n_epochs` = 344. The `surprise.SVD` baseline uses parameters `lr_all` = 0.005, `reg_all` = 0.100, `n_factors` = 100, `n_epochs` = 100. SVD++ with augmented data uses `lr_all` = 0.003, `reg_all` = 0.100, `n_factors` = 100, and `n_epochs` = 200. The `surprise.KNNBaseline` model was trained with parameters `k` = 50, `sim_options` = {`name` = `pearson_baseline`, `user_based` = False, `shrinkage` = 20}, and `bsl_options` = {`method` = `als`, `n_epochs` = 20, `reg` = 0.02}.

Note that "SVD" (no ++) was included to justify using SVD++ over just using SVD, since it did not perform as well. Also, the augmented data in "SVD++ with augmented data" was an attempt to improve Problem 2's model by using extra data about the users. That is, we grouped users of the same age and gender (i.e. whether or not they were male). Finally, "KNNBaseline" is another model from surprise, that uses KNN and a bias for the dataset to create embeddings.

(i) We found that our leaderboard MAE was about 1.55% lower than the test set MAE in Problem 2. To study this difference, we compared the mean number of ratings per user and per item across the two sets. In the test set, users had an average of 95.432 ratings and items had 53.503 ratings, while in the leaderboard set, users had 181.245 ratings on average and items had 149.821 ratings. Since both metrics are substantially higher in the leaderboard set, it is clearly more dense than the test set.

With a denser dataset, the implicit feedback in SVD++ becomes more informative, because users and items have more interactions contributing to their embeddings. The term in SVD++ that incorporates implicit feedback has a stronger effect, nudging user embeddings closer to their true preferences. As a result, the SVD++ model achieves better performance on the leaderboard set compared to the test set.

(ii) Interestingly, our more simple LF model (from Problem 1) outperformed our chosen Problem 2 method in MAE on the same testing split. This implies that the LF model outperforms our SVD++ model, at least in sparser data sets, like the development set. Note that, since we did not assess our Problem 1 model with the leaderboard test set, we do not know its performance. It would be useful to, as our leaderboard performance was significantly better than our test set for all submissions of Problem 2's model to the leaderboard. Since SVD++ is able to handle sparse data well, unlike a standard LF model, it is possible that our Problem 1 solution would not see an increase in performance on the leaderboard set like our problem 2 solution.

2d

Our current approach (SVD++) with a randomized search for hyperparameter configurations is robust, but still has some areas for improvement. As mentioned in 2a, an SVD++ implementation can be more robust than the already solid SVD algorithm. However, its main drawback is the time it takes to train its models, especially when combined with good practices like a hyperparameter search over broad ranges of values. To combat this, we chose a RandomizedSearchCV, and for this reason, our process yields models faster, since we search 20 configurations instead of 96, at a slight cost to peak performance. Given our understanding that our model works better on users and items that have high numbers of reviews (see 2c(i)), our model would likely perform worse on extremely sparse datasets. For users with *no ratings*, the model would perform terribly, as it would predict that, for any movie, the new user would give a rating that is exactly the global mean rating of the train set. Therefore, this model would work poorly for introducing users to a movie service without a warm start (i.e. the new user tells the service some of the movies they like).

An improvement that can be made is in our data representation scheme. In the present scheme, we represent each user and each item separately, which is sensitive to variability in the data. Also, according to the difference in our leaderboard score and test score, we've seen that our model will favor users and movies with more ratings. So, by aggregating ratings into a superuser, grouped based on common demographics (like gender, age), we could create more dense ratings per user. This makes the data more dense, and therefore a trained model more performant.

A great lesson we learned in this project is the importance of plots to represent data. We found it very helpful to plot validation and training loss across many epochs, as we found it easy to determining underfitting or overfitting easily. This was helpful as we were deciding ranges of hyperparameters, since we could surmise if we wanted to scale up or down in magnitude. For instance, a very slow trace plot indicates learning rate needs to increase. We also found it helpful to inspect the data. In particular, recording measures of the train, validation, and test sets helped us justify performance among the datasets. Without this, we would not have been able to inspect the difference in sparsity between the development test set and the leaderboard test set, and improve on the model in Problem 2. Finally, we learned a great lesson in the importance of randomized search, which often gave us slightly worse results than an exhaustive grid search, but also gave us feedback far faster than one. Because of the quicker turnaround time, we were able to generate plots, adjust our hyperparameters, and run another iteration of randomized search much quicker. So, we were able to get a performant model much faster.