CSC311H, Fall 2024
Final Project

Addison Luo (1009405948) + Erika Liang (1008938107)

Part A:

1. **Solution:**

   (a) Using the values of k = [1, 6, 11, 16, 21, 26]

```
k = 1, Validation Accuracy: 0.6245
k = 6, Validation Accuracy: 0.6781
k = 11, Validation Accuracy: 0.6895
k = 16, Validation Accuracy: 0.6756
k = 21, Validation Accuracy: 0.6692
k = 26, Validation Accuracy: 0.6523
```

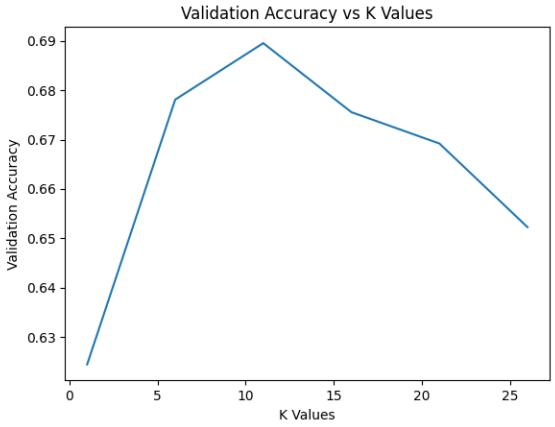

Figure 1: Graph of Validation Values vs. K Values

   (b) Based on the results of knn.py, the $k*$ value with the highest performance is $k = 11$ with a validation accuracy of 0.6895. The final overall test accuracy is 0.6842.

   (c) (a)
```
k = 1, Validation Accuracy: 0.6071
k = 6, Validation Accuracy: 0.6542
k = 11, Validation Accuracy: 0.6826
k = 16, Validation Accuracy: 0.6860
k = 21, Validation Accuracy: 0.6922
k = 26, Validation Accuracy: 0.6904
Best k: 21 with validation accuracy: 0.6922
```

   (b) Based on the results, the $k*$ value with the highest performance is $k = 21$ with a validation accuracy of 0.6922. The final overall test accuracy is 0.6816.

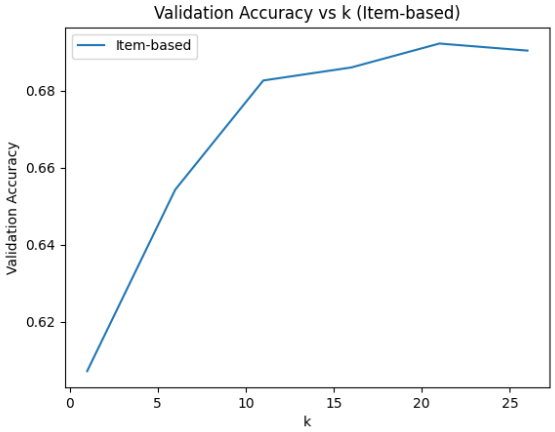

Figure 2: Graph of Validation Values vs. K Values (Item-based)

(d) When the two methods are compared, the resulting validation accuracies of the item-based method are slightly higher. However, the user-based method yields a slightly higher test accuracy than the item-based method. Because test accuracy provides an unbiased estimate of how well the model performs on data it has not seen before, it simulates real-world scenarios and gives a more accurate reflection of how the model will behave when deployed. Thus, although both methods have similar performance, the method that performs better is the one with the higher testing accuracy, the user-based method.

(e) There are several limitations one might face while approaching K-Nearest Neighbours. For instance, in this traditional KNN algorithm, every neighbour within the k-neighbourhood contributes equally to predictions, even if some are farther away and less relevant. This can introduce noise, especially when datasets become more sparse or high-dimensional. is that of scalability. As the number of user/items increases in a KNN algorithm, it becomes more computationally expensive as the computing distances between all pairs of users increase. This can lead to slower predictions once the databases become large.

2. **Solution:**

(a) The likelihood that question j is correctly answered by student i is given by:

$$p(c_{ij} = 1|\theta_i, \beta_j) = \frac{exp(\theta_i - \beta_j)}{1 + exp(\theta_i - \beta_j)}$$

$$p(c_{ij}|\theta_i, \beta_j) = (\frac{exp(\theta_i - \beta_j)}{1 + exp(\theta_i - \beta_j)})^{c_{ij}}(1 - \frac{exp(\theta_i - \beta_j)}{1 + exp(\theta_i - \beta_j)})^{1-c_{ij}}$$

Then, the likelihood for all students and all questions would be written as the product of the individual likelihoods. To derive the log likelihood of the probability, we need to get the sum of the log probabilities of each observation. Thus, we get:

$$\log p(C|\theta_i, \beta_j) = \sum_{i,j} \log[(\frac{exp(\theta_i - \beta_j)}{1 + exp(\theta_i - \beta_j)})^{c_{ij}}(1 - \frac{exp(\theta_i - \beta_j)}{1 + exp(\theta_i - \beta_j)})^{1-c_{ij}}]$$

$$\log p(C|\theta_i, \beta_) = \sum_{i,j} c_{ij} \log(\frac{exp(\theta_i - \beta_j)}{1 + exp(\theta_i - \beta_j)}) + (1 - c_{ij}) \log(1 - \frac{exp(\theta_i - \beta_j)}{1 + exp(\theta_i - \beta_j)})$$

Looking at the given likelihood, we can see that it is actually a form of the sigmoid function such that:

$$\sigma(x) = \frac{exp(x)}{1 + exp(x)}$$

We can rewrite the log equation as such:

$$\log p(C|\theta_i, \beta_j) = \sum_{i,j} [c_{ij} \log(\sigma(\theta_i - \beta_j)) + (1 - c_{ij}) \log(1 - \sigma(\theta_i - \beta_j))]$$

Log of the sigmoid function and its complement can be written as such:

$$\log(\sigma(\theta_i - \beta_j)) = \theta_i - \beta_j - \log(1 + exp(\theta_i - \beta_j))$$

$$\log(1 - \sigma(\theta_i - \beta_j)) = -\log(1 + exp(\theta_i - \beta_j))$$

Substituting these values in, we get:

$$\log p(C|\theta_i, \beta_j) = \sum_{i,j} [c_{ij}(\theta_i - \beta_j - log(1 + exp(\theta_i - \beta_j))) + (c_{ij} - 1) \log(1 + exp(\theta_i - \beta_j))]$$

The final simplified expression for log-likelihood is:

$$\log p(C|\theta_i, \beta_j) = \sum_{i,j} [c_{ij}(\theta_i - \beta_j) - \log(1 + exp(\theta_i - \beta_j))]$$

The derivative with respect to θ_i is:

$$\frac{\partial}{\partial \theta_i} \log p(C|\theta, \beta) = \sum_{j} [c_{ij} - \frac{exp(\theta_i - \beta_j)}{1 + exp(\theta_i - \beta_j)}]$$

The derivative with respect to β_j is:

$$\frac{\partial}{\partial \theta_i} \log p(C|\theta, \beta) = -\sum_{i} [c_{ij} - \frac{exp(\theta_i - \beta_j)}{1 + exp(\theta_i - \beta_j)}]$$

(b) The parameters I selected was a learning rate of 0.01 and 1000 iterations. The training curve that shows the training and validation log-likelihoods as a function of iteration is as follows:

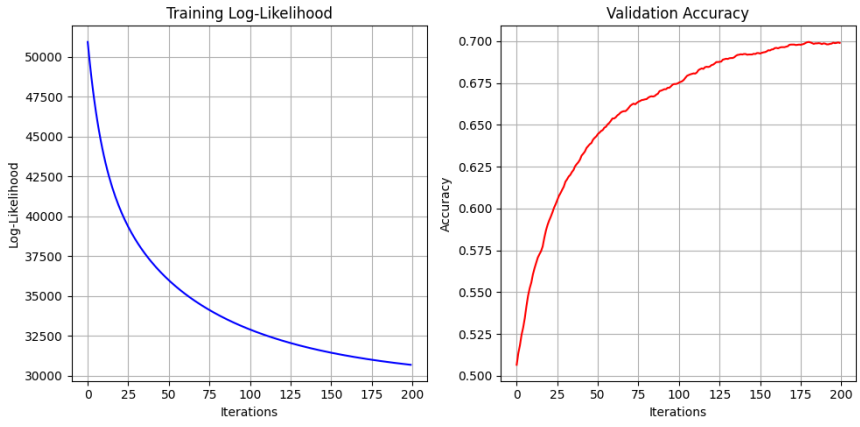

Figure 3: Training Curve of the Training and Validation Log-Likelihoods as a Function of Iteration

(c) `Final Validation Accuracy: 0.6977`
`Final Test Accuracy: 0.7025`

(d) I selected question 0, 10, and 20. All of the resulting curves are sigmoid-shaped (s-shaped). These curves represent the probability of answering a specific question correctly based on student ability.

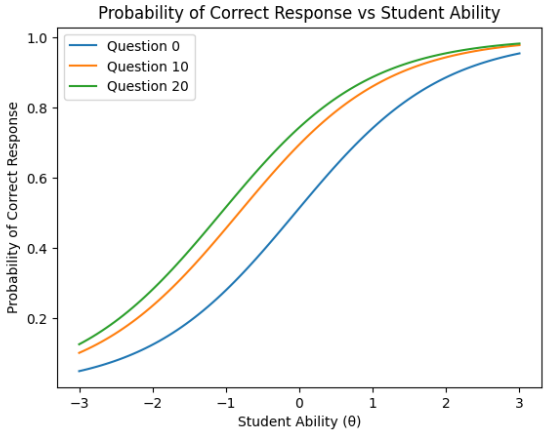

Figure 4: Plot of the Probability of Correct Response vs. Student Ability

3. **Solution:**

Option 1: Matrix Factorization

(a) We will try values of $k = [2, 5, 10, 20, 50]$. Here are the reported results:

```
Starting SVD reconstruction for different k values...
k = 2, Validation Accuracy = 0.6579
k = 5, Validation Accuracy = 0.6590
k = 10, Validation Accuracy = 0.6586
k = 20, Validation Accuracy = 0.6540
k = 50, Validation Accuracy = 0.6485

Best k = 5, Best Validation Accuracy = 0.6590
Test Accuracy with best k = 5: 0.6636
```

(b) One limitation of SVD is that the missing entries in the matrix are filled with the mean of each question's response, which may not accurately represent the true values of the missing entries. Filling missing entries with a mean value ignores potential reasons/patterns behind why the entries were missing.

(c) When we use SGD, we choose a single point in the dataset at each iteration to compute the gradient. We can compute the gradient by taking the derivative of the objective function w.r.t. $\mathbf{u}_n$ and $\mathbf{z}_m$. Here is the derivation of the gradient rules:
Focusing on the (n, m)-th term, the loss function is:

$$
\begin{aligned}
L_{nm} &= \frac{1}{2}\left(C_{nm} - \mathbf{u}_n^\top \mathbf{z}_m\right)^2 \\
\frac{\partial L_{nm}}{\partial \mathbf{u}_n} &= \frac{\partial}{\partial \mathbf{u}_n}\frac{1}{2}\left(C_{nm} - \mathbf{u}_n^\top \mathbf{z}_m\right)^2 \\
&= (C_{nm} - \mathbf{u}_n^\top \mathbf{z}_m) \cdot \frac{\partial}{\partial \mathbf{u}_n}(-\mathbf{u}_n^\top \mathbf{z}_m) \\
&= -(C_{nm} - \mathbf{u}_n^\top \mathbf{z}_m) \cdot (\mathbf{z}_m) \\
&= -\text{ error } \cdot \mathbf{z}_m
\end{aligned}
$$

Symmetrically, we have:

$$
\frac{\partial L_{nm}}{\partial \mathbf{z}_m} = -\text{ error } \cdot \mathbf{u}_n
$$

(d) For hyperparameters, we adjusted the number of iterations first, and then chose the learning rate that gave the best results.

```
learning_rate = 0.06
num_iterations = 70000
```

We will try values of $k = [2, 5, 10, 20, 50]$ again. Here are the results:

```
Starting ALS with SGD for different k values...
k = 2, Final Validation Accuracy = 0.6850
k = 5, Final Validation Accuracy = 0.6884
k = 10, Final Validation Accuracy = 0.6854
k = 20, Final Validation Accuracy = 0.6933
k = 50, Final Validation Accuracy = 0.6897

Best k for ALS = 20, Best Validation Accuracy = 0.6933
Test Accuracy (ALS) with best k = 20: 0.6926
```

(e) See Figure 5. This is the plot of training and validation squared-error losses. Notice that the losses of the training data are much higher than the validation. This is because the training dataset is a larger percentage of the overall data than the validation set and the `square_error_loss()` function does not return losses scaled by the number of data points.

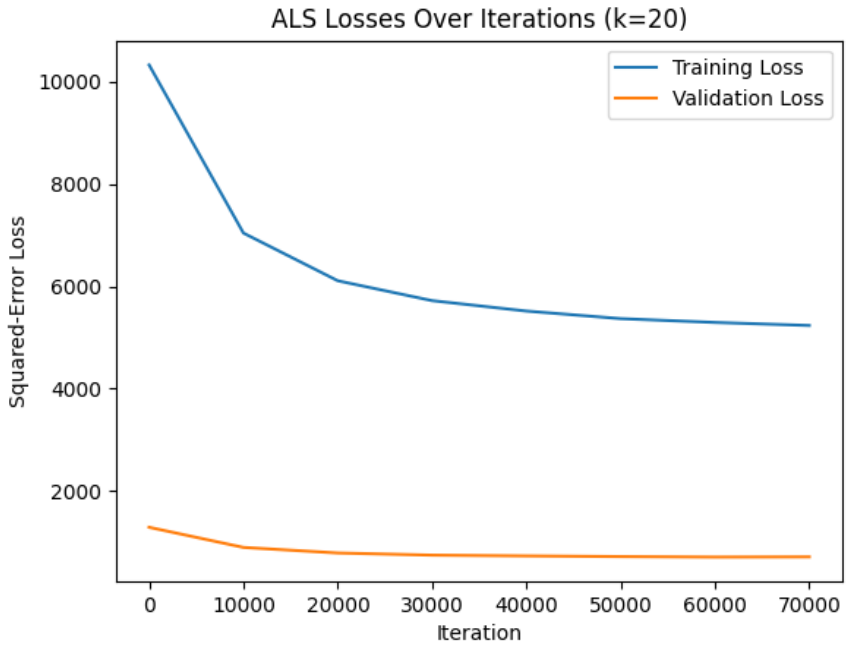

Figure 5: ALS Squared-Error Losses over Iterations, $k* = 20$

4. **Solution:**

See `ensemble.py`.

In this question, we will be implementing a bagging ensemble of models, using decision trees as our base model.

We generate three different bootstrapped samples from the training data. Each bootstrapped sample is created by randomly sampling the data with replacement, which results in some data points being

duplicated while others might not appear at all. This helps reduce variance by introducing some diversity in the training set for each model.

Each of these bootstrapped training sets is used to train a base model. In our case, we will be using the DecisionTreeClassifier decision tree from SKLearn. The decision trees are trained with different random seeds to ensure variability between them.

After training the three decision trees, we make predictions for the validation and test sets. For each data point, the ensemble predicts the average of the three model predictions. The predicted correctness is averaged, and the final decision is made by thresholding the average at 0.5, i.e., if the average prediction is greater than or equal to 0.5, the ensemble predicts 1 (correct); otherwise, it predicts 0 (incorrect).

The ensemble's performance is evaluated by computing its accuracy on both the validation and test sets. In the main function, we compare the ensemble's accuracy to the accuracy of individual base models to determine if the ensemble improves over the base models.

In the first implementation of the ensemble, we noticed that the ensemble predicted very well on the training set, but the accuracy of the validation accuracy significantly dropped. This signalled to us that there was a high likelihood that the decision trees were overfitting to the training data and not generalizing well to unseen data.

```
Ensemble Training Accuracy: 0.9078
Ensemble Validation Accuracy: 0.6041
Ensemble Test Accuracy: 0.5893
```

Given the evidence of overfitting, we decided to focus on tuning the `max_depth` hyperparameter of the decision trees. The `max_depth` parameter controls how deep the tree can grow before it stops splitting. When the depth is too large, the tree has too many levels and becomes overly specific to the training data. Reducing the depth of the tree can help mitigate overfitting by forcing the tree to learn broader, more generalizable patterns. However, if the tree depth is too shallow, we would see the opposite and the tree would be more likely to underfit and not learn enough about the data.

Thus, we tested different tree depths (e.g., 3, 5, 10, 15, 20). This process allowed us to identify the optimal tree depth, and we landed on `max_depth` = 15. Here are the results:

```
Starting hyperparameter tuning for max_depth...
Testing max_depth = 3
Validation Accuracy = 0.6142
Testing max_depth = 5
Validation Accuracy = 0.6269
Testing max_depth = 10
Validation Accuracy = 0.6455
Testing max_depth = 15
Validation Accuracy = 0.6506
Testing max_depth = 20
Validation Accuracy = 0.6351

Best max_depth = 15, Best Validation Accuracy = 0.6506

Evaluating ensemble with best max_depth...
Final Validation Accuracies for each individual model: [0.6335, 0.6435, 0.6351]
Final Validation Accuracy for Ensemble = 0.6506
Final Test Accuracy = 0.6475
```

We can see that the ensemble has a higher validation accuracy by about 1-2% higher than each individual decision tree. This is because a single decision tree is likely to have high variance due to

overfitting the training data. By averaging the predictions of several decision trees trained on different bootstrapped samples, bagging helps reduce this variance. This results in a more stable model that generalizes unseen data better.

Part B:

We chose to expand upon the K-Nearest Neighbours (KNN) algorithm. In Part A, one of the factors we found that limited the performance of this algorithm was that every neighbour within the k-neighbourhood contributes equally to predictions, regardless of their distance or contextual relevance. This uniformity can introduce noise because data points that are farther away and less relevant can influence predictions just as much as closer, more relevant ones. In the context of predicting student responses, this approach can often fail to capture factors such as common student misconceptions, different question difficulties, and the temporal relevance of past questions. In order to address this, we propose extending the traditional KNN algorithm by adding a weighted system, where neighbours can contribute to predictions based on different weighing schemes. With this modification, we can aim to improve prediction accuracy by capturing real-life nuances of the problem domain. For our weighing schemes, we will focusing on 2 types of weighting: distance-based and feature-based.

We have implemented the distance-based weighting algorithm by increasing the contribution of closer neighbours so that neighbours contribute inversely based on their distance to the query point. This addresses a key limitation of the original KNN algorithm where neighbours are all treated equally, and instead, neighbours can now had different effects based on relevant proximity. Ideally, given these changes, students or questions with similar attributes are more likely to provide accurate predictions in its domain. We expect these changes to improve optimization by reducing noise from distant points. Additionally, we expect emphasizing closer points to smoothen predictions. There are many different approaches to the weighing algorithm which assigns weights to different extents. We have implemented 3 kinds: Using inverse distance, the inverse square distance, and Gaussian bell curve weighting.

Inverse distance weighting assigns weight such that closer neighbours contribute significantly more than farther ones, giving this weight algorithm:

$$w_i = \frac{1}{d_i + \epsilon}$$

where the weight of the i-th neighbour (w_i) is equal to the reciprocal of the distance of the i-th neighbour (d_i) plus a small constant to avoid zero division (ϵ).

Inverse square distance weighting assigns even greater weights to closer neighbours. It has the same formula as inverse distance weighting but the denominator is squared:

$$w_i = \frac{1}{(d_i + \epsilon)^2}$$

Gaussian bell curve weighting uses a bell curve function to assign weights, leading to exponentially higher weights being given to closer points. The formula for this weighting is:

$$w_i = e^{-\frac{d_i^2}{2\sigma^2}}$$

where σ is the bandwidth parameter controlling the influence of distance.

For value $x_{i,j}$, we can predict it by using the average of the weighted k nearest neighbours, where w_n is the weight of neighbour n, calculated using the weighting schemes mentioned above:

$$x_{ij} = \frac{\sum_{n \in N_k(i,j)} w_n * x_n}{\sum_{n \in N_k(i,j)} w_n}$$

To approach the algorithm through code and find the efficacy of our weighted approach, we wanted to compute the distances between the target and all other rows to reflect user-based and item-based calculations. Then, after selecting k nearest neighbours for select k values, we looked to applied the chosen weighing schemes in order to predict the missing value using the weighted average of neighbours. However, as we attempted to evaluate the accuracy on validation and test datasets, we found that evaluating even one k value took an extremely long time. There are many factors that could lead to this, but most likely, this approach exacerbated the second issue we found about knn: scalability. Distance-based weight algorithms rely on the calculation of pairwise distances, which can involve costly operations. While weights can often be derived in advance, the approach of distance-based requires a calculation to be made at each point. These real-time calculation can add to computational burden, and is the most likely reason it was difficult to get results in a timely manner with a dataset as large as the one we were given. However, focusing on columns over rows (which can have many more dimensions) can decrease computational overhead. We decided to try feature-based weighting systems, which through relying on direct feature values, can skip costly pairwise operations.

We have implemented the feature-based weighting algorithm by incorporating metadata into the similarity computation, allowing neighbours to contribute based on both their response data and additional contextual information. This addresses a key limitation of the original KNN algorithm, which relied solely on raw response similarity, by leveraging metadata to refine the relationships between students or questions. Additionally, our approach will attempt mitigate the curse of dimensionality by projecting the data onto a lower-dimensional manifold defined by meaningful metadata attributes, such as shared subject tags for questions or demographic information for students. By preserving important patterns and connections within the data, we aim to assign weights that better reflect meaningful proximities. Neighbours with similar attributes are thus more likely to contribute accurate predictions within their domain. We expect this approach to improve optimization by reducing noise from irrelevant neighbours and enhancing the reliability of predictions, while also capturing nuanced relationships beyond raw response data.

The feature-based weighting KNN was implemented in two configurations:

- User-based similarity: Neighbours are selected based on student-level similarities. We used categories (each with equal weighting) in the `student_meta.csv` file to produce a similarity score:

$$\texttt{student\_sim\_score} = \frac{\texttt{gender\_sim} + \texttt{premium\_sim} + \texttt{age\_sim}}{3}$$

  Since in the dataset, gender and premium were binary variables, two students would have a score of 1 for each if they matched, and 0 if they didn't. For age, a smaller age difference yielded a higher similarity.

$$\texttt{age\_sim} = \frac{1}{1 + |s1_{age} - s2_{age}|}$$

- Item-based similarity: Neighbours are selected based on question-level similarities. We use the Jaccard similarity from set theory. For questions $q1$ and $q2$, we find the sets containing every question with the same subject as $q1$ and then $q2$. Then, the similarity is the size of the intersection of the sets, over the size of the union of the 2 sets. In our case, we have:

$$\texttt{question\_sim\_score} = \frac{|q1_{subject} \cap q2_{subject}|}{|q1_{subject} \cup q2_{subject}|}$$

Then, we compute a similarity matrix, to which we will refer to when taking the k-nearest-neighbours so we will not need to continuously compute similarities between the same points.

Experimenting with the same k values as hyperparameters, here are the results for a smaller 500 by 500 sparse matrix:

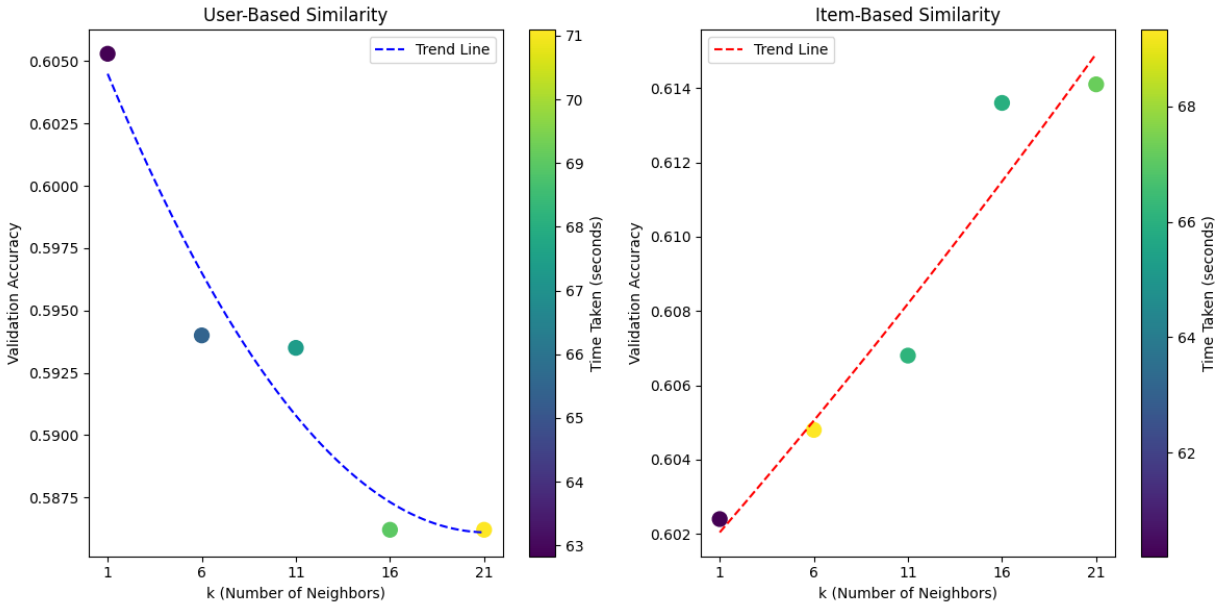

Figure 6: KNN k vs Validation vs Time for User/Item Based Similarity for a smaller dataset

Overall, we see that the feature-weighted KNN algorithm performs worse than the original KNN implemented in part A.

K-Value	Original KNN Accuracy	Feature-Weighted KNN Accuracy	Time Taken (s)
1	0.6245	0.6053	62.83
6	0.6781	0.5940	65.46
11	0.6895	0.5935	67.35
16	0.6756	0.5862	69.02
21	0.6692	0.5862	71.09

Table 1: User-Based Comparison with Time Taken for Weighted KNN

K-Value	Original KNN Accuracy	Feature-Weighted KNN Accuracy	Time Taken (s)
1	0.6071	0.6024	60.20
6	0.6542	0.6048	69.33
11	0.6826	0.6068	66.14
16	0.6860	0.6136	65.95
21	0.6922	0.6141	67.25

Table 2: Item-Based Comparison with Time Taken for Weighted KNN

An explanation for the worse performance could be that feature-based weighting introduces more complexity to the model and potentially more noise as well. While the idea is to emphasize the most relevant features, it might not always align with the true importance of features in making predictions. The weights might not sufficiently distinguish between meaningful and irrelevant features, for example, we weighed all of gender, age, and premium equally when considering the similarities between users, but this may not be an accurate representation of the true weights. This could have lead to the overfitting of the data to features that did not matter. We can observe that our results show that as k increases, validation accuracy decrease for user-based similarity weighting and accuracy increases for item-based similarity.

The decrease in accuracy for user-based could be explained by noise with larger k. As we increase k, the model incorporates more neighbours into the decision, which may dilute the effect of closer, more relevant neighbours. If the additional neighbours are not truly similar to the query point, this can reduce the accuracy of predictions. For example, our similarity score using the metadata may not accurately predict similarity between users. In our implementation, we used metadata (such as gender, age, and premium pupil status) to calculate similarity between users. This approach assumes that users who share similar demographic characteristics (e.g., age, gender, and socio-economic status) will behave similarly in terms of answering questions. However, this assumption has its limitations. For example, age may provide a rough indication of a student's educational level, but it does not account for individual differences prior knowledge or academic interests/involvement. Thus, these factors alone do not fully capture the multifaceted nature of student learning. Using these factors to calculate similarity might not always lead to accurate predictions of how students will perform on diagnostic questions.

On the other hand, for item-based similarity, the trend of increasing accuracy with larger k suggests that question relationships are better captured with a broader set of similar questions, and the model benefits from considering more neighbours. Item-based similarity relies on the assumption that questions have inherent similarities based on the subjects they belong to.

A limitation of our approach of feature-based weights in KNN is the computational cost associated with calculating the nearest neighbours, especially when working with large datasets. Specifically, the process of computing similarities between users or items is computationally intensive, as it involves pairwise comparisons across potentially large matrices. For example, during testing, we tried using the feature-KNN algorithm for the whole sparse matrix and received these results using cProfile:

```
Running feature-weighted KNN (k = 5) with user-based similarity...
Validation Accuracy: 0.5967
...
519654132 function calls (519646597 primitive calls) in 3370.999 seconds
```

The algorithm took 56 minutes to return, even after optimizations like identifying bottlenecks and instead, precomputing important objects like similarity matrix and global means. Thus, we had to use a smaller dataset for demonstration, including the first 500 by 500 entries in the sparse matrix. We used cProfile and discovered most of the overhead was now in sorting the similarity scores to take the top k:

```
Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   233510   49.375    0.000   65.470    0.000 {built-in method builtins.sorted}
```

To help mitigate these computational challenges, some solutions that could be implemented could be parallelization and attempting to reduce dimensionality. Distributing the computations across multiple processors or using GPU-based acceleration could help reduce the overall computation time. Reducing the number of features used in the similarity calculations (via techniques like PCA) could help reduce both computational cost and mitigate the curse of dimensionality.