CSC311 Final Project Report

Charles Cheung, Jarrett Jia, Matthew Lack

August 2024

1 Preamble

2 Part A

2.1 k-nearest neighbors

2.1.1 Part A

The main() function in knn.py was implemented by iterating over the selection of k-values $= \{1, 6, 11, 16, 21, 26\}$ and recording the output of knn_impute_by_user() on each k-value to the terminal. The results were first reported separately to the terminal (Log 2.1.5), then plotted as the blue graph in (Figure 1).

Note that the logging report to the terminal, as well as the validation accuracy plot per k-value, have been condensed together with later work, such as parts B, C, and D for brevity.

2.1.2 Part B

As reported by the logging report at (Log 2.1.5), the best performing k-value for kNN imputation by user was $k = 11$ with a final validation accuracy of 68.92464031547841%. Then, the test performance of the kNN model with $k^* = k = 11$ was 68.21902346247473%.

2.1.3 Part C

knn_impute_by_item() was implemented similarly to how knn_impute_by_user() was implemented given the starter code. The main difference is contrary to the line in starter code in knn_impute_by_user():

$$\text{mat} = \text{nbrs.fit\_transform(matrix)}$$

In knn_impute_by_item(), we instead perform:

$$\text{mat} = \text{nbrs.fit\_transform(matrix.T).T}$$

This is such that we instead perform the .fit_transform() operation on a per-user basis instead of how it would have been applied per-question if the transpose operation were left out.

This is, of course, assuming that similar items (questions) will have similar patterns of responses from users. This means that the missing values for a particular item can be accurately imputed based on the values of other similar items.

Repeating parts A and B, we output the logging report (Log 2.1.5) to the terminal and plot the information as the orange graph in (Figure 1). The k-value with the best final validation accuracy was $k = 26$ with a validation accuracy of 69.17866215071973%. Then, the test performance of the kNN model with $k^* = k = 26$ was 68.30369743155518%

2.1.4 Part D

There is a small margin of 0.1% between the test accuracies of user- and item-based collaborative filtering. The item-based collaborative filtering kNN model performs marginally better at 68.30369743155518% test accuracy.

[TODO discussion of results?]

2.1.5 Part E

1. The kNN computation itself does not scale well, and suffers from the Curse of Dimensionality quite badly. In our task as a proof of concept, this program runs well due to the minimal size that we need to deal with. When scaling to the entire online education platform's (Eedi's) dataset, with more features (questions or students depending on imputation direction), this program would not run nearly as fast because distance between datapoints in the feature space becomes less meaningful.

2. The kNN model could suffer from imputation bias where there could be some relationship or pattern in the data that could not be represented as distance.

3. kNN is a local algorithm. If there were global patterns, our model would not be able to act upon them.

Logging Report to the Terminal

```
Sparse matrix:
[[nan nan nan ... nan nan nan]
 [nan 0.  nan ... nan nan nan]
 [nan nan 1.  ... nan nan nan]
 ...
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]
```

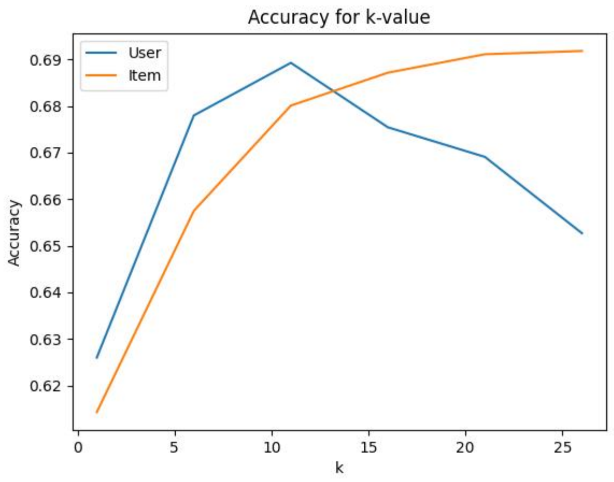


Figure 1: kNN Model Final Validation Accuracy per k-value

```
 [nan nan nan ... nan nan nan]]
Shape of sparse matrix:
(542, 1774)
Valid acc of user with k = 1: 0.6206231442280553
Valid acc of user with k = 6: 0.6779565340107254
Valid acc of user with k = 11: 0.6892464031547841
Valid acc of user with k = 16: 0.6754613385833121
Valid acc of user with k = 21: 0.669056737647772792
Valid acc of user with k = 26: 0.652695455828394
Valid acc of item with k = 1: 0.6143909068855944
Valid acc of item with k = 6: 0.657493649446919
Valid acc of item with k = 11: 0.6800733841377364
Valid acc of item with k = 16: 0.6871295512277731
Valid acc of item with k = 21: 0.6918010450791973
Valid acc of item with k = 26: 0.6917866215071973
User’s best k* is 11 with valid acc 0.6892464031547841
Item’s best kk* is 26 with valid acc 0.6917866215071973
Test acc of chosen user k* = 11: 0.6821902346247473
Test acc of chosen item kk* = 26: 0.6830369743155518
```

2.2 Item Response Theory

2.2.1 Part C

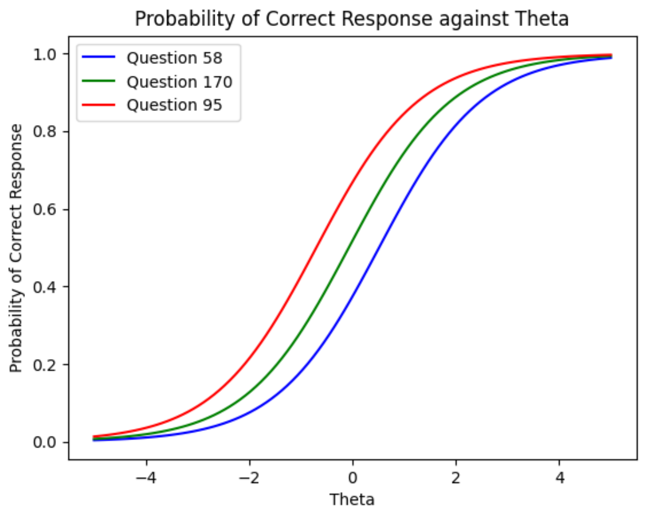
The final validation accuracy was 70.80158058142817% with test accuracy of 69.74315551792266%.

2.2.2 Part D

To get the best view of the change in probability across questions, we took questions $j_1 = 58, j_2 = 170, j_3 = 95$ respectively representing approximately one of the lowest, middle and highest possible probabilities of questions.

The questions j_n were selected after reviewing which questions of index between 0 and 175 were suitable to showcase a good difference in possible probabilities. We acknowledge that this is a completely subjective metric and that the team member was too bothered to review the probabilities of the other questions with index from 175 to 1773; however we believe that the chosen j_n suitably represent the example differential that we desired to show.

To represent the j^{th} question, we took the corresponding β_j of the now-fixed j, and plotted its sigmoid$(\theta - \beta)$ probability over θ. The plot can be found at (Figure **??**).



Probability of Correct Response on Fixed Questions against Theta

The curves differ in height most prominently around $\theta = 0$.

2.3 Neural Networks

[TODO check out CUDA, Charles doesnt have nvidia]

2.3.1 Part A

1. ALS is generally deterministic given a fixed number of iterations, as it alternates between optimizing two sets of parameters at a time. Neural networks are not limited to be deterministic, noted by employing Stochastic

Gradient Descent and randomly initialized weights, or in more advanced neural nets, Dropout techniques or random seeding.

2. ALS only works with data that can be represented as matrices, such as small images or sparse matrices. On the other hand, neural networks can handle any data that can be squashed as a 1-dimensional vector, like text, images, and in advanced NNs like CNNs and RNNs, timeseries datasets.

3. ALS can only capture linear relations with its simple linear algebra, but neural nets can capture non-linear relationships with activation functions.

4. It is easy to see the individual relationship between inputs and outputs with an ALS since outputs are directly decomposed from the input data, thus it is easy to see which factors correspond to whichever features in the input data. On the other hand, neural networks make this hard to see, as they have complex layers, non-linear activation functions, and the most probable case of neural nets storing a layer's feature information in multiple of the layer's nodes at once, instead of just one node of the layer.

5. ALS are generally cheaper to run, as they only solve linear equations and perform matrix multiplication. Neural networks are much slower as they perform matrix multiplication on vectors of much larger dimensions, as well as performing backpropagation which is twice as slow as a matrix multiplication (forward pass).

2.3.2 Part B

The AutoEncoder class was implemented by:

- Using the starter code for __init__() and get_weight_norm()
- For .forward(), simply using the torch.sigmoid() function after applying the g layer, and after applying the h layer onto the result of the previous.
- Allowing the remaining methods to be given by the parent class torch.nn.Module.

2.3.3 Part C

With Learning Rate = 0.005 and 75 epochs, it was observed that the AutoEncoder model set with $k = 50$ had the highest validation accuracy at68.79762912785775% and final training cost of 7878.5.

Our methodology for tuning both hyperparameters was to try commonly seen hyperparameter values seen in other neural network projects, in powers of 10 or other rounded numbers. For example, test hyperparameter values for learning rate were $0.1, 0.01$ and 0.001, and for epochs, $10, 25, 50, 100$ and 200.

To justify our hyperparameter choices of $LR = 0.005$ and 75 epochs, we noted that:

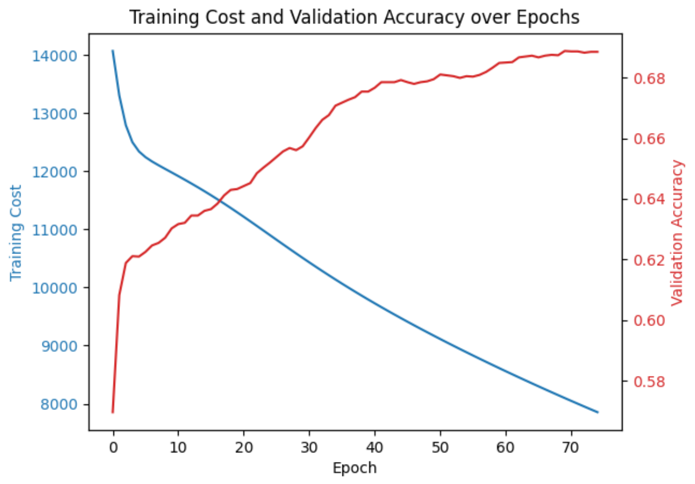


Figure 2: Training Cost and Validation Accuracy over Epochs

- Increasing the number of epochs above 75 allowed the validation accuracy to stagnate and flatline at the cost of computation time. This happened particularly for higher values of k, such as $k = 200$ or $k = 500$.
- Decreasing the number of epochs below 75 did not allow for enough time for the validation accuracy to rise, for all k-values.
- Increasing the learning rate above 0.005 certainly helped the models of all k-values to learn faster, however at the expense of much larger divergence and unlearning when nearing the convergence point (we found that validation accuracies converged to just below 69% in all our tuning, for all k-values).
- Decreasing the learning rate below 0.005 did not result in any meaningful learning. Models learned too slowly and compensating with an increased number of epochs meant inefficient use of computing power and time.

2.3.4 Part D

The chosen $k^* = 50$ with hyperparameters $LR = 0.005$ and 75 epochs performed with a test accuracy of 68.55771944679651%.

The plot graphing training cost and validation accuracy as a function of epochs is located at (Figure 2).

During the training process, there was a sharp drop in training cost, indicating that the model found the fast way down. Then, as the validation accuracy stopped growing quickly, the changes in training cost became approximately linear, decreasing with respect to epoch.

On the other hand, the validation accuracy sharply rose, then approximately matched a shallow logarithmic shape until its plateau near the end of its training arc.

Then, our analysis agrees with our hyperparameter choice in Part C. We avoided underfitting by allowing the validation accuracy to improve, but we also successfully avoided overfitting by stopping training just as validation accuracy started to plateau but not too late such that training cost continued to decrease while validation accuracy did not further increase.

2.3.5 Part E

With the L_2 regularized loss added onto the hyperparameters of the previously found $k^* = 50$, $\lambda = 0.001$ performed the best out of the allowed choices $\{0.001, 0.01, 0.1, 1\}$.

- Higher values of λ decreased final validation and test accuracy.
- On the same hyperparameters chosen in part D,
  - $\lambda = 0.001$ had a validation accuracy of 68.75529212531752% and a test accuracy of 68.47304544171606%.
  - $\lambda = 0.01$ had a validation accuracy of 68.06378775049393% and a test accuracy of 67.99322607959356%.
  - $\lambda = 0.1$ had a validation accuracy of 62.60231442280553% and a test accuracy of 62.54586508608524%.
  - $\lambda = 1$ had a validation accuracy of 62.58820208862546% and a test accuracy of 62.23539373412362%.

The training logging was still active and training progress and epoch validation accuracy was available to see during the training process. During the training for the two higher values of λ, the validation accuracy was mostly monotonic, starting around 55%, rising to and plateauing to approximately 62.5% accuracy. This indicates that the model never overshot any local minima. In turn, this means that:

1. The models with higher values of λ were underfitting and not learning the small details well, and the accuracy plateau near 69% achieved by the models with lower values of λ was desirable with respect to the model's generalization behavior.
2. It would not be correct to surmise that the models with lower values of λ, or without L_2 loss, were overfitting and that the accuracy plateau near 62.5% achieved by the models with higher values of λ was desirable with respect to the model's generalization behavior. This is because the validation accuracy of the models with L_2 regularized loss generally matched with their respective test accuracies; meaning that the models did not overfit with respect to the validation datasets. [TODO more analysis needed here]

For the best performing $\lambda = 0.001$ compared to the results of k^* with its validation accuracy in part C and its test accuracy in part D, there was no significant difference in final validation and test accuracy, aside from marginally worse scores. This is an acceptable outcome, as the closer the value of $\lambda \to 0$, the less difference there should be to SSE loss function.

2.4 Ensemble

To improve the stability and accuracy of our base models, 3 base neural network AutoEncoder models were selected.

2.4.1 Implementation

To implement bagging ensembling, the averaged naive models' prediction was calculated first by taking the np.mean of the collection of their final validation and test accuracies.

Then, the bagging began. For each bag for a preset NUM_BAGS (odd number), a bootstrap of the training matrix was taken by np.random.choice, $n =$ size(train_matrix.shape) times each with replacement.

A new AutoEncoder model was then initialized for each bootstrapped training set and run, with the tuning from previous sections: $k = 50$, $lr = 0.005$, $\lambda = 0.001$ for 75 epochs.

One meaningful change was to modify the evaluation function to take the majority vote of the models during evaluation over the ensemble, instead of taking the direct output of a single model. Then using this function, the ensemble was evaluated for final validation and test accuracy.

Finally, the two sets of accuracies were returned and logged to the terminal.

2.4.2 Results

For Bag Size = 3:

- The averaged correctness prediction from the three base neural network models returned a naive validation accuracy of 68.51067833286293% and a naive test accuracy of 68.22843164926145%.
- In contrast, the bagged ensemble of three models returned a validation accuracy of 67.2876093705899% and a test accuracy of 67.54163138583121%.

Thus, there is a marginal decrease in general accuracy after bagging. This is seen by the decrease in both validation and testing, and not by only one or the other.

[TODO] This is contrary to what we expect from bagging ensembling. Theoretically, bagging should provide a decrease in prediction variance as a trade-off for increased running time of multiple models, and does not mean a small decrease in performance. To investigate this behavior, the bag size was increased to different levels.

For Bag Size = 7:

- The averaged correctness prediction from the seven base neural network models returned a naive validation accuracy of 68.82114968482455% and a naive test accuracy of 68.66120989745036%.
- In contrast, the bagged ensemble of seven models returned a validation accuracy of 66.56788032740615% and a test accuracy of 66.72311600338696%.

For Bag Size = 15:

- The averaged correctness prediction from the fifteen base neural network models returned a naive validation accuracy of 68.69413867720388% and a naive test accuracy of 68.10612475303416%.
- In contrast, the bagged ensemble of fifteen models returned a validation accuracy of 67.6545300592718% and a test accuracy of 67.93677674287327%.

For Bag Size = 31:

- The averaged correctness prediction from the thirty-one base neural network models returned a naive validation accuracy of 68.45893310753599% and a naive test accuracy of 68.29428920876847%.
- In contrast, the bagged ensemble of thirty-one models returned a validation accuracy of 67.79565340107254% and a test accuracy of 67.76742873271239%.
- The total program running time was very long and painful.

It seems like the trend of averaged naive models versus the bagged ensemble of models is that they are roughly equal in terms of predicted correctness / accuracy, minus some small margin. There was no performance increase in terms of prediction accuracy. This was to be expected, as bagging ensembles generally do not provide direct correctness increases, as they only provide decreases in prediction variance instead. However, the small margin in the decrease of accuracy between averaged naive models versus the bagged ensemble of models was mysterious. [TODO need more analysis here!!!]

3 Part B

3.1 Introductory Analysis

Analyzing the current models, it is clear that accuracy plateaus just shy of 70%. Since this effect appears for all tested models, it is safe to assume that it is a limitation of the dataset rather than the specific algorithm used. Despite this, we experimented with many tweaks to the existing algorithms attempting to use the same data, steering clear of the metadata. The following methods were attempted:

1. An embedded KNN, where an autoencoder is pre-trained on the data, and then the latent representations are used as embeddings for the KNN to impute values.

2. Using scores calculated using the covariance matrix of the dataset. Each entry would be the dot product of the rows in the covariance matrix corresponding to the questions the user has answered, with -1 representing a wrong answer and 1 representing a correct answer. Then, using an autoencoder

3. Clustering questions by subject using the question metadata, and including a "user embedding" in the input to the autoencoder network, the embedding being an average of the most correctly answered subjects.

4. Adding drop-out layers to the autoencoder.

5. Increasing the amount of layers in the autoencoder.

6. Using different activations in the deep autoencoder models.

7. Adding normalized representations of user metadata to the input of the autoencoder.

8. Different schedules for the learning rate.

3.1.1 Embedded KNN

The idea behind this algorithm is to train an autoencoder to create rich low-dimensional representations of the user data, then use these representations to improve the KNN imputation method by reducing the effect of the curse of dimension and removing unused information from the user representations. This idea did not hold up in practice however, and reduced the accuracy of the KNN from 65% to 62% at best, even when weighting the nearest neighbors inversely by distance.

3.1.2 Covariance Matrix

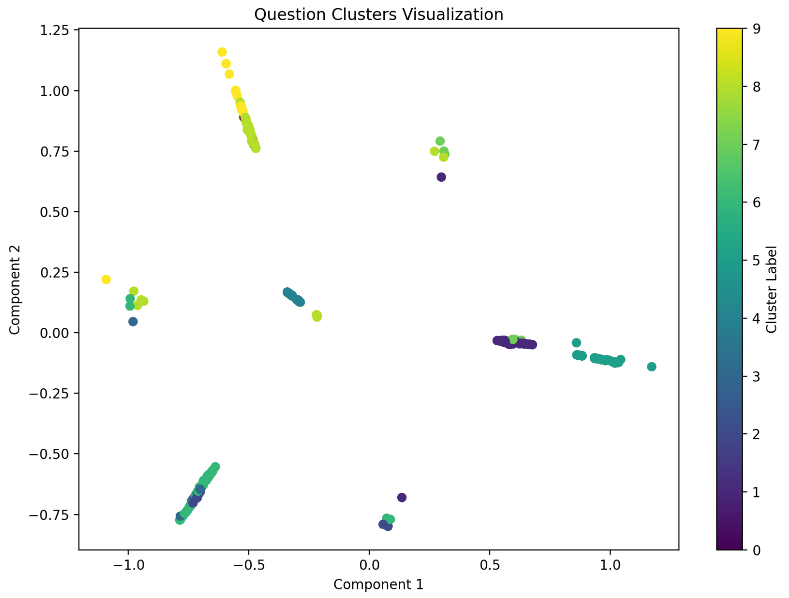
This algorithm is not an extension of a previous algorithm but a side-quest. The idea behind this algorithm is to create a covariance matrix where each entry describes the correlation between a pair of questions, then to use this matrix to calculate the probability that some question would be answered correctly by a user given the user's existing answers. Given a user's observed answers $u \in \{-1, 0, 1\}^{1774}$, where $u_i = 0$ if the data is missing, and a query question i, we have:

$$\text{score}_i = u \cdot \Sigma_i$$

Where Σ_i if the i^{th} row (or column) in the covariance matrix of the data. If the score is above 0, then we count that as a positive guess, otherwise it is a negative guess. This method achieves a test accuracy of 66.4%.

3.1.3 Clustering

We clustered the questions together based on the subjects they belong to, such a clustering looks as such when projected to 2D:



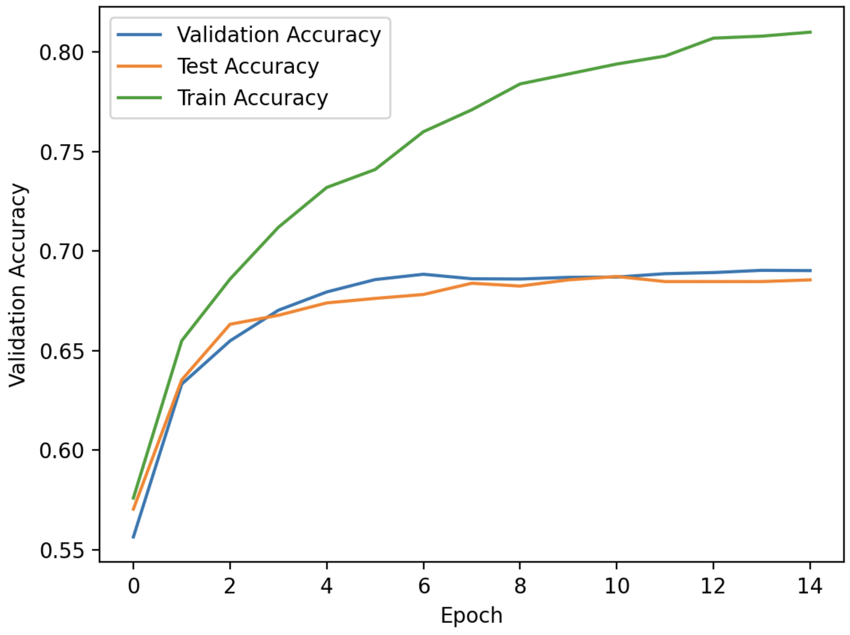
We used K-Means clustering with $k = 10$, reducing the number of effective "subjects" down to 10. Performance seemed to be best with this value. Other values there were tried are $k \in \{10, 20, 25, 50, 100\}$. Next, we used this clustering to provide user embeddings to supply to an autoencoder as extra features. The embeddings were calculated as the sum of the one-hot-encoded representations of the clusters of each question the user had answered correctly. Say Q is the set of questions a user answered correctly, and for $q \in Q$, q_e is the one-hot vector representing the cluster q belongs to. The user embedding is calculated as:

$$\sum_{q \in Q} q_e$$

and is concatenated to the end of the input vector for the autoencoder. With this method, we achieved a maximum test accuracy of 68.32%, out of the different values of k we ran it with.

3.1.4 Drop out layers

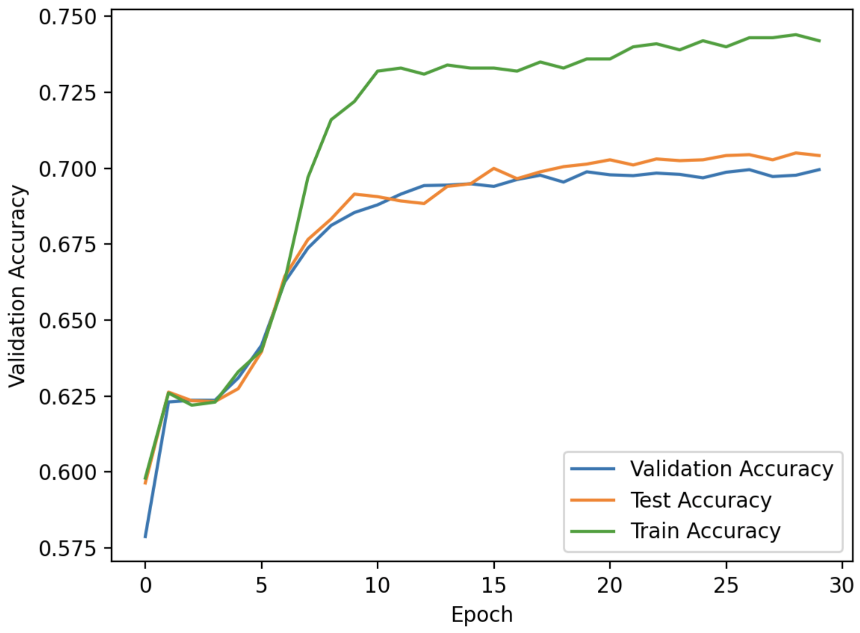
Did not notably change results. This is surprising given that drop-out layers is a type of regularization and is supposed to reduce overfitting, which is somewhat visible in a graph of the accuracy of the vanilla autoencoder.



Note that epochs in the graph are scaled down by a factor of 5, so on the graph epoch 6 = epoch 30.

3.1.5 Deep Auto Encoder (Proposed Method)

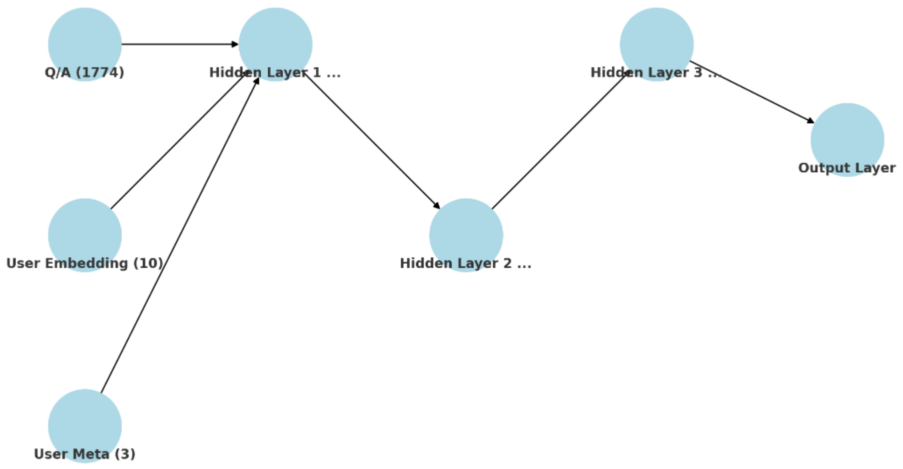
This was done by adding two more layers around the middle layer of the autoencoder, both the same size of k, this allows for more complex transformations to the latent space representation. This resulted in an interesting graph of the accuracies:



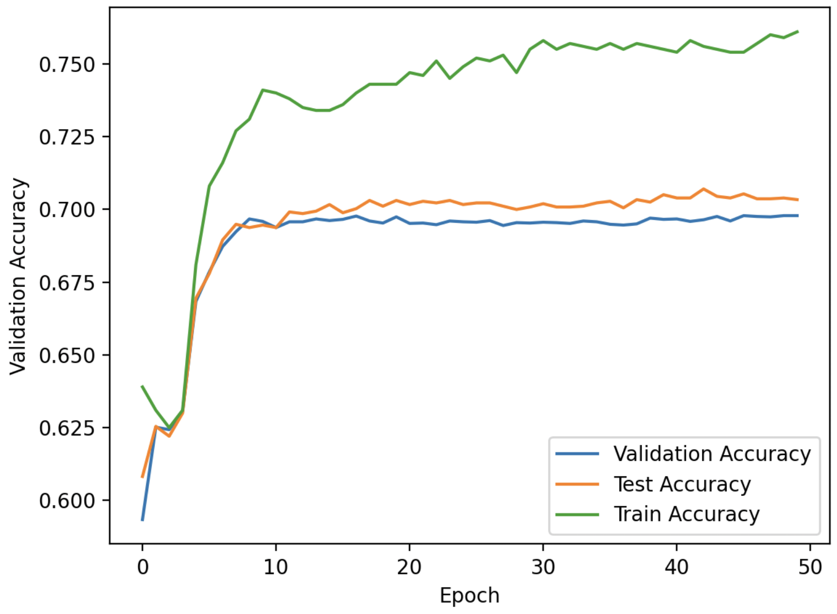
This resulting in accuracies close to 70%, and when combined with these next two modifications, we were able to get 70.4% test accuracy on the data. The two modifications we made were both examples of feature engineering. They are intended to give the model more information to go off of as it seems from analysis that there are too many questions and far too many unanswered questions or missing data. We included a user embedding calculated the same as was described in the **Clustering** section above, as well as one more 3-dimensional input, being a normalized and vectorized version of the user metadata for the user.

- The user's birth date is used to calculate their age and then normalized by dividing by 100.
- The user's gender is given as a number just as it is in the csv.
- The user's premium pupil status is given either as 0, or as given in the csv

A diagram of the proposed mode is given below:



And below is the training graph:



Reaching a maximum test accuracy of 70.4%

3.2 Comparison

3.3 Limitations

Some limitations of this algorithm are as follows

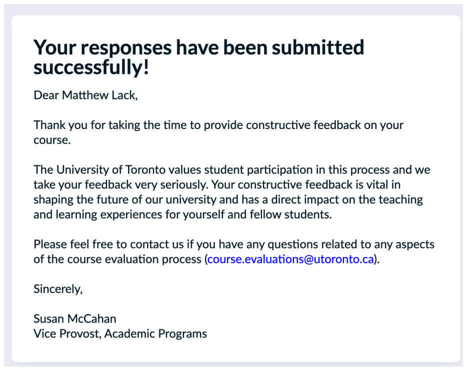
Model/Modification	**Test Accuracy (%)**
kNN User-Based (k=11)	68.21
kNN Item-Based (k=26)	68.30
Autoencoder (k=50)	68.55
Autoencoder + L2 Regularization (λ=0.001)	68.47
Bagging Ensemble (Bag Size=3)	67.54
Bagging Ensemble (Bag Size=7)	66.72
Bagging Ensemble (Bag Size=15)	67.93
Bagging Ensemble (Bag Size=31)	67.77
Embedded KNN	62
Clustering	68.32
Covariance Matrix	66.4
Deep Autoencoder + Clustering + User Embedding	70.4

Table 1: Test Accuracy of Different Models and Modifications

- This method will perform poorly or at the level of the vanilla autoencoder if there is no correlation between the questions, or between the user's age, financial situation, or gender and the answers.
- Only so much information is available in the provided dataset and it may not be possible to get higher accuracies because there may just not be enough information available. The dataset is very sparse and there is not much to go off of. The accuracy plateau around 70% hints at this.
- This model does not take into account the names of the subjects, we could fix this if we were to use publicly available textual embeddings. We might be able to better cluster the questions and subjects.
- This autoencoder has more layers and is more computationally expensive to train than the vanilla autoencoder.
- (Personal Intuition) It seems to me that since the objective of the autoencoder is to reconstruct its input, that using it to fill in this many missing values is not viable since there are more missing values than present values, the model may just focus on capturing the patterns of missing values, of which there may not even be any.

4 Confirmation of Course Evaluations

We assert that all of us have completed our course evaluations.



Your responses have been submitted successfully!

Dear Matthew Lack,

Thank you for taking the time to provide constructive feedback on your course.

The University of Toronto values student participation in this process and we take your feedback very seriously. Your constructive feedback is vital in shaping the future of our university and has a direct impact on the teaching and learning experiences for yourself and fellow students.

Please feel free to contact us if you have any questions related to any aspects of the course evaluation process (course.evaluations@utoronto.ca).

Sincerely,

Susan McCahan
Vice Provost, Academic Programs

5 Contributions

Matthew Lack - Part B model experimentation and writeup. Binhe Jia - kNN and IRT Charles - Neural Nets and Ensembling

6 LLMs

Charles, GPT4o: "How do I use matplotlib with Python if I have two graphs (differing axis ranges of course) that I want to put on the same matplotlib graph?"

Matthew Lack (All GPT-4o):

- Chat Log 1
- Chat Log 2
- Chat Log 3
- Chat Log 4