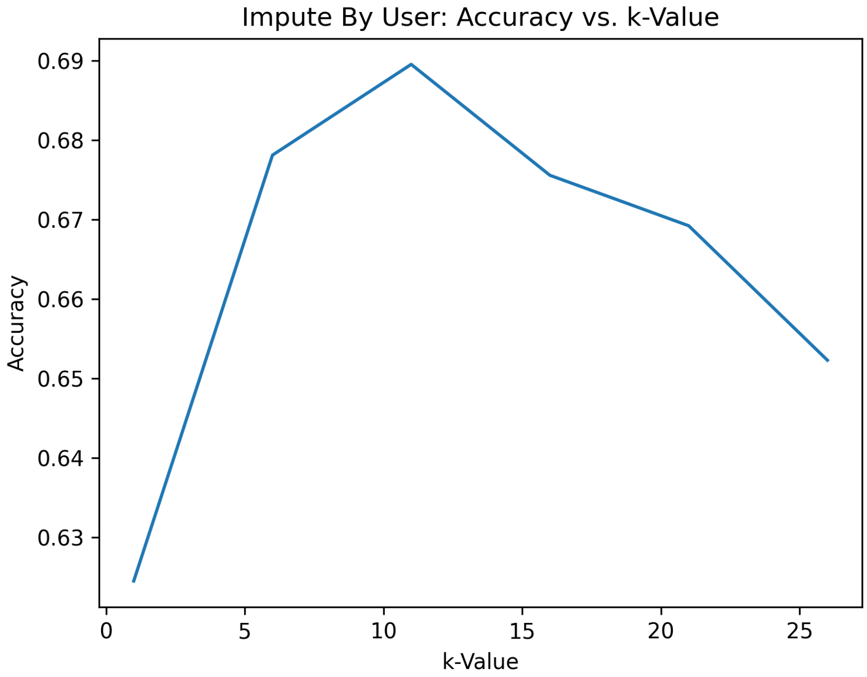
CSC311 Project Report

Edward Li, Haojun Qiu, Jenci Wei

Fall 2022

Part A

Question 1a



```
Impute By User: k-value: 1, Accuracy: 0.6244707874682472
Impute By User: k-value: 6, Accuracy: 0.6780976573525261
Impute By User: k-value: 11, Accuracy: 0.6895286480383855
Impute By User: k-value: 16, Accuracy: 0.6755574372001129
Impute By User: k-value: 21, Accuracy: 0.6692068868190799
Impute By User: k-value: 26, Accuracy: 0.6522720858029918
```

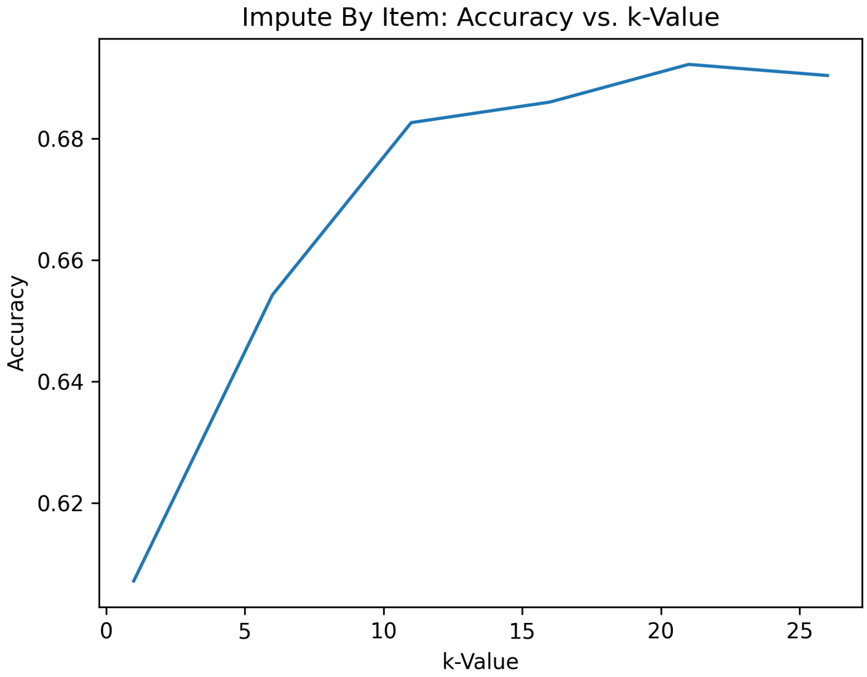
Question 1b

The k^* that has the highest performance on validation data is $k^* = 11$

```
k_star: 11, Test Accuracy: 0.6841659610499576
```

Question 1c

The underlying assumption on item-based collaborative filtering is that if question A is answered correctly/incorrectly in a similar way as question B from students that commonly answer both questions, then whether question A is correctly answered by specific students matches that of question B.



```
Impute By Item: k-value: 1, Accuracy: 0.6244707874682472
Impute By Item: k-value: 6, Accuracy: 0.6780976573525261
Impute By Item: k-value: 11, Accuracy: 0.6895286480383855
Impute By Item: k-value: 16, Accuracy: 0.6755574372001129
Impute By Item: k-value: 21, Accuracy: 0.6692068868190799
Impute By Item: k-value: 26, Accuracy: 0.6522720858029918
```

The k^* that has the highest performance on validation data is $k^* = 21$

```
k_star: 21, Test Accuracy: 0.6683601467682755
```

Question 1d

Based on the above results, imputing by user performs better on the test dataset.

Question 1e

1. kNN is computationally expensive at test time, therefore with a larger dataset, it may become computationally infeasible.

2. With a large dataset, it is expensive to store the model (i.e. one would have to store the entire dataset). On contrary, some parametric model only requires storing a fixed amount of parameters that does not scaled with the size of the datset.

3. This method highly relies on the quality of existing data points. Since the data point is sparse — e.g., when imputing by question, only a small common subset of students have answered two questions so each point (student) is heavily weighted during nearest neighbour finding (similarly when imputing by user). If the existing data is noisy, the predictions would be of low accuracy.

Question 2a

Let N_s and N_q denote the number of students and the number of questions, separately. And hence, the two vector parameters are denoted as

$$\begin{aligned}\boldsymbol{\theta} &= (\theta_1, \theta_2, \ldots, \theta_{N_s}) \,,\\ \boldsymbol{\beta} &= \left(\beta_1, \beta_2, \ldots, \beta_{N_q}\right) .\end{aligned}$$

Next, let's make two important observations on independence between variables.

(I) For all $i \in \{1, \ldots, N_s\}$, the i-th student's joint probability of correctly answering all N_q questions, are independent of which of all other students, conditioning on the ability and difficulty parameters — since each depends only on its own (not others') ability parameter θ_i.

(II) For a *fixed* $i \in \{1, \ldots, N_s\}$ (i.e., fix a specific student), for all $j \in \{1, \ldots, N_q\}$ (i.e., for any questions j), the probability of student i correctly answering questions j are independent of which of any other questions j', conditioning on the ability and difficulty parameters — since it only depends on θ_i and β_j (not $\beta_{j'}$ for any $j' \neq j$).

The above two will result in variables $\{c_{ij}\}_{i,j}$ being mutually independent, conditioning on the parameters $\boldsymbol{\theta}, \boldsymbol{\beta}$. Let's first note that for any $i \in \{1, 2, \ldots, N_s\}$, $j \in \{1, 2, \ldots, N_q\}$,

$$\begin{aligned}p(c_{ij} \,|\, \theta_i, \beta_j) &= \left(\frac{\exp(\theta_i - \beta_j)}{1 + \exp(\theta_i - \beta_j)}\right)^{c_{ij}} \left(1 - \frac{\exp(\theta_i - \beta_j)}{1 + \exp(\theta_i - \beta_j)}\right)^{1-c_{ij}}\\ &= \left(\frac{\exp(\theta_i - \beta_j)}{1 + \exp(\theta_i - \beta_j)}\right)^{c_{ij}} \left(\frac{1}{1 + \exp(\theta_i - \beta_j)}\right)^{1-c_{ij}} .\end{aligned}$$

Then, observe that the likelihood function is

$$\begin{aligned}L(\boldsymbol{\theta}, \boldsymbol{\beta}) &= p(\mathbf{C} \,|\, \boldsymbol{\theta}, \boldsymbol{\beta})\\ &= \prod_{i=1}^{N_s} p(c_{i,1}, c_{i,2}, \ldots, c_{i,N_q} \,|\, \boldsymbol{\theta}, \boldsymbol{\beta}) = \prod_{i=1}^{N_s} p(c_{i,1}, c_{i,2}, \ldots, c_{i,N_q} \,|\, \theta_i, \boldsymbol{\beta}) && \text{(by observation (I))}\\ &= \prod_{i=1}^{N_s} \left(\prod_{j=1}^{N_q} p\left(c_{ij} \,|\, \theta_i, \boldsymbol{\beta}\right)\right) = \prod_{i=1}^{N_s} \prod_{j=1}^{N_q} p\left(c_{ij} \,|\, \theta_i, \beta_j\right) && \text{(by observation (II))}\\ &= \prod_{i=1}^{N_s} \prod_{j=1}^{N_q} \left(\frac{\exp(\theta_i - \beta_j)}{1 + \exp(\theta_i - \beta_j)}\right)^{c_{ij}} \left(\frac{1}{1 + \exp(\theta_i - \beta_j)}\right)^{1-c_{ij}} .\end{aligned}$$

Hence, the log-likelihood function is

$$\begin{aligned}\ell(\boldsymbol{\theta}, \boldsymbol{\beta}) = \log L(\boldsymbol{\theta}, \boldsymbol{\beta}) &= \log \left(\prod_{i=1}^{N_s} \prod_{j=1}^{N_q} \left(\frac{\exp(\theta_i - \beta_j)}{1 + \exp(\theta_i - \beta_j)}\right)^{c_{ij}} \left(\frac{1}{1 + \exp(\theta_i - \beta_j)}\right)^{1-c_{ij}}\right)\\ &= \sum_{i=1}^{N_s} \sum_{j=1}^{N_q} \log \left(\left(\frac{\exp(\theta_i - \beta_j)}{1 + \exp(\theta_i - \beta_j)}\right)^{c_{ij}} \left(\frac{1}{1 + \exp(\theta_i - \beta_j)}\right)^{1-c_{ij}}\right)\\ &= \sum_{i=1}^{N_s} \sum_{j=1}^{N_q} \left(c_{ij} \log \left(\frac{\exp(\theta_i - \beta_j)}{1 + \exp(\theta_i - \beta_j)}\right) + (1 - c_{ij}) \log \left(\frac{1}{1 + \exp(\theta_i - \beta_j)}\right)\right)\\ &= \sum_{i=1}^{N_s} \sum_{j=1}^{N_q} \Big(c_{ij}\Big((\theta_i - \beta_j) - \log(1 + \exp(\theta_i - \beta_j))\Big) + (c_{ij} - 1) \log(1 + \exp(\theta_i - \beta_j))\Big)\\ &= \sum_{i=1}^{N_s} \sum_{j=1}^{N_q} \Big(c_{ij}(\theta_i - \beta_j) - \log(1 + \exp(\theta_i - \beta_j))\Big).\end{aligned}$$

Next, let's derive for the partial derivative of log-likelihood w.r.t. its parameters.

- For all $i \in \{1, 2, \dots, N_s\}$, we have that

$$\begin{aligned}
\frac{\partial}{\partial \theta_i} \ell(\boldsymbol{\theta}, \boldsymbol{\beta}) &= \frac{\partial}{\partial \theta_i} \left(\sum_{i'=1}^{N_s} \sum_{j=1}^{N_q} \Big(c_{i'j}(\theta_{i'} - \beta_j) - \log\left(1 + \exp\left(\theta_{i'} - \beta_j\right)\right) \Big) \right) \\
&= \frac{\partial}{\partial \theta_i} \left(\sum_{j=1}^{N_q} \Big(c_{ij}(\theta_i - \beta_j) - \log\left(1 + \exp\left(\theta_i - \beta_j\right)\right) \Big) \right) \\
&= \sum_{j=1}^{N_q} \left(c_{ij} - \left(\frac{1}{1 + \exp\left(\theta_i - \beta_j\right)} \cdot \exp\left(\theta_i - \beta_j\right) \cdot (1) \right) \right) \\
&= \sum_{j=1}^{N_q} \left(c_{ij} - \frac{\exp\left(\theta_i - \beta_j\right)}{1 + \exp\left(\theta_i - \beta_j\right)} \right) \\
&= \sum_{j=1}^{N_q} \left(c_{ij} - \sigma(\theta_i - \beta_j) \right),
\end{aligned}$$

  where $\sigma(.)$ is the sigmoid function.

- For all $j \in \{1, 2, \dots, N_q\}$, we have that

$$\begin{aligned}
\frac{\partial}{\partial \beta_j} \ell(\boldsymbol{\theta}, \boldsymbol{\beta}) &= \frac{\partial}{\partial \beta_j} \left(\sum_{i'=1}^{N_s} \sum_{j=1}^{N_q} \Big(c_{i'j}(\theta_{i'} - \beta_j) - \log\left(1 + \exp\left(\theta_{i'} - \beta_j\right)\right) \Big) \right) \\
&= \frac{\partial}{\partial \beta_j} \left(\sum_{j=1}^{N_q} \Big(c_{ij}(\theta_i - \beta_j) - \log\left(1 + \exp\left(\theta_i - \beta_j\right)\right) \Big) \right) \\
&= \sum_{j=1}^{N_q} \left(-c_{ij} - \left(\frac{1}{1 + \exp\left(\theta_i - \beta_j\right)} \cdot \exp\left(\theta_i - \beta_j\right) \cdot (-1) \right) \right) \\
&= \sum_{j=1}^{N_q} \left(-c_{ij} + \frac{\exp\left(\theta_i - \beta_j\right)}{1 + \exp\left(\theta_i - \beta_j\right)} \right) \\
&= \sum_{j=1}^{N_q} \left(-c_{ij} + \sigma(\theta_i - \beta_j) \right) \\
&= \sum_{j=1}^{N_q} - \left(c_{ij} - \sigma(\theta_i - \beta_j) \right),
\end{aligned}$$

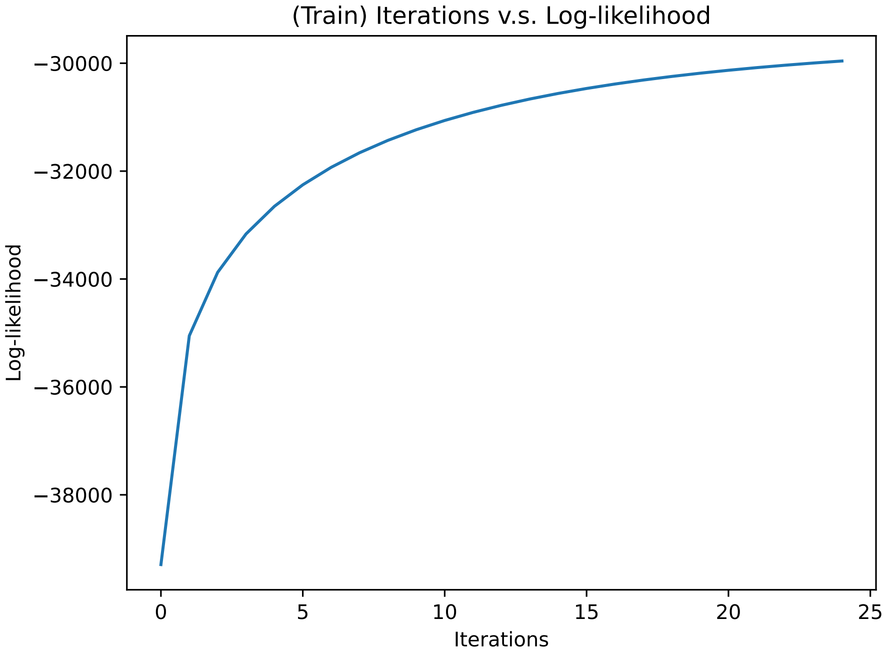
  where $\sigma(.)$ is the sigmoid function.

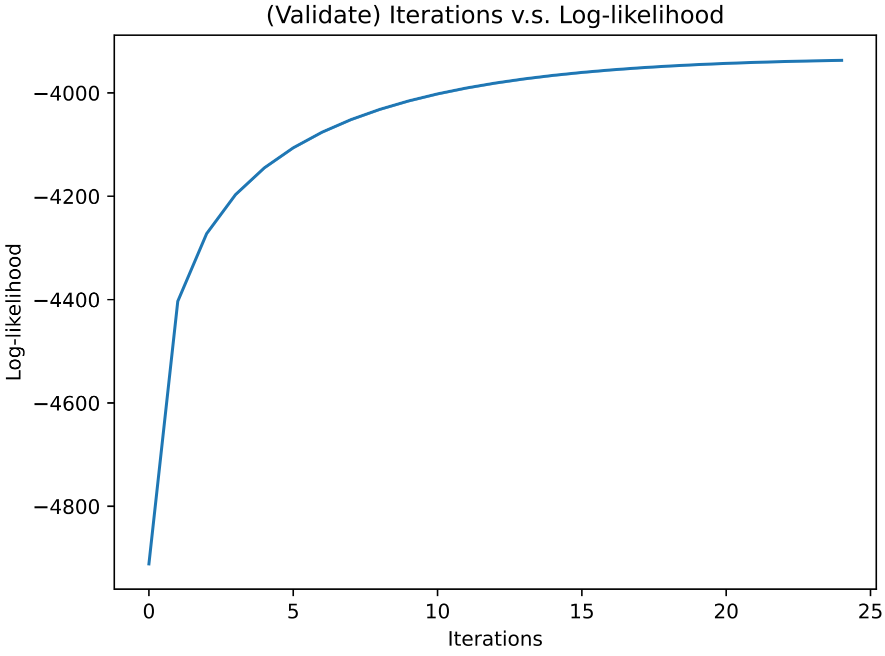
Question 2b

Hyper-parameters selected (based on validation likelihood curve):

Learning Rate	0.01
# Iteration	25

Training & validation log-likelihood:



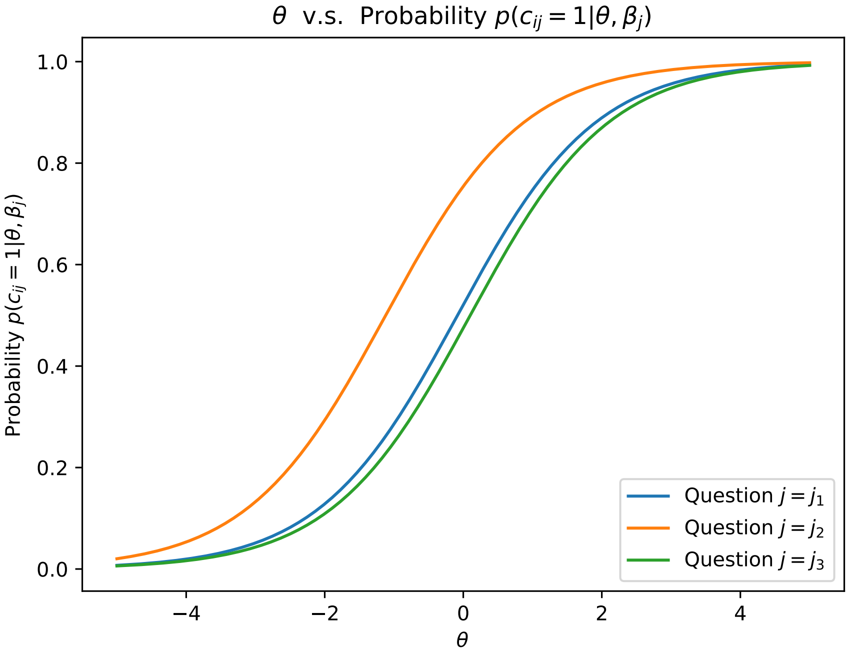


Question 2c

Final validation & test accuracies:

	Validation	Test
Accuracy	0.7065	0.7053

Question 2d



We manually chose three questions $j_1 = 10$, $j_2 = 20$, $j_3 = 500$. All three curves have a shape that looks like sigmoid function. Let's fix $j \in \{j_1, j_2, j_3\}$. This result is not surprising as the value on the y-axis represents the probability term

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

as a function of θ. And this is a shifted-sigmoid function

$$f(\theta) = p(c_{ij} = 1 \,|\, \theta, \beta_j) = \sigma(\theta - \beta_j).$$

To interpret the meaning of the curves, it basically tells that for a fix question j (hence an associated fixed question-difficulty β_j), the probability of a student answering that question j correctly, as a function of its ability θ, is monotonic (and more specific, grows like a sigmoid function). I.e., the higher the ability of a student, the higher the chance that student will answering that question j correctly. This matches our intuition, and telling that the IRT model is not non-sense or made up arbitrarily.

Question 3a (Neural Networks)

The three key differences are as follows:

1. ALS is a linear model while neural network can be linear or nonlinear. More specifically, ALS, as a linear model, has smaller hypothesis space than the auto-encoder, that is, auto-encoder could have been trained to represent much more complex function (with regard to linearity) compared to the ALS.

2. ALS is more interpretable than neural network. For example, for a latent dimension of k, we could possibly learn a latent space where each dimension represents a subject content. More specifically, we could possibly learn to state where the first dimension represent subject "machine learning", and the first dimension of $\mathbf{u}_n \in \mathbb{R}^k$ represents the capability of user n at machine learning problems, while the first dimension of $\mathbf{z}_m$ represents how much the question m relates to "machine learning". However, neural network, specifically an autoencoder, aims to find the best mapping from input to its latent and then to its reconstruction, and the whole process is less or more like a blackbox. Although for a trained model, we eventually get its weights and bias, and all the latent representation of the input, but all of which are hard to interpret (e.g., it may be hard to know what's the first latent dimension of a latent space means, ever).

3. ALS and neural network use different optimization method. For ALS, we optimize matrices U and Z by solving linear equations iteratively. However, neural networks typically use gradient descent since they are usually nonlinear models, rather than via any types of written-out analytic equations.

Question 3b (Neural Networks)

Please check our neural_network.py

Question 3c (Neural Networks)

As shown in our neural_network.py, We use three nested for loop to find the best k, learning rate, and number of epoch. It turns out that the best set of parameters is $k = 10$, learning rate = 0.1, and number of epoch = 15, which reaches the best validation accuracy of 0.6843.

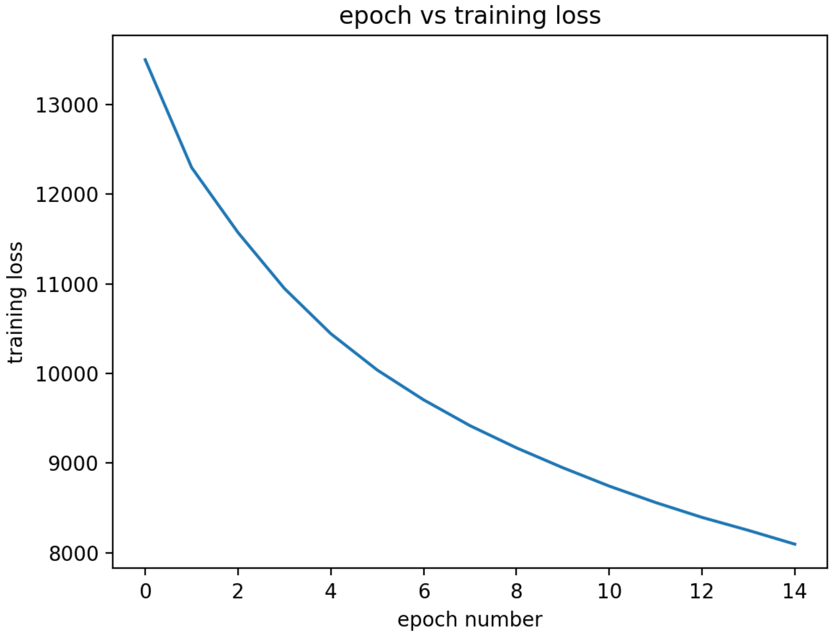
Parameter Name	Value
k	10
Learning Rate	0.1
# epoch	15

Hence, $\boxed{k^* \text{ is } 10}$.

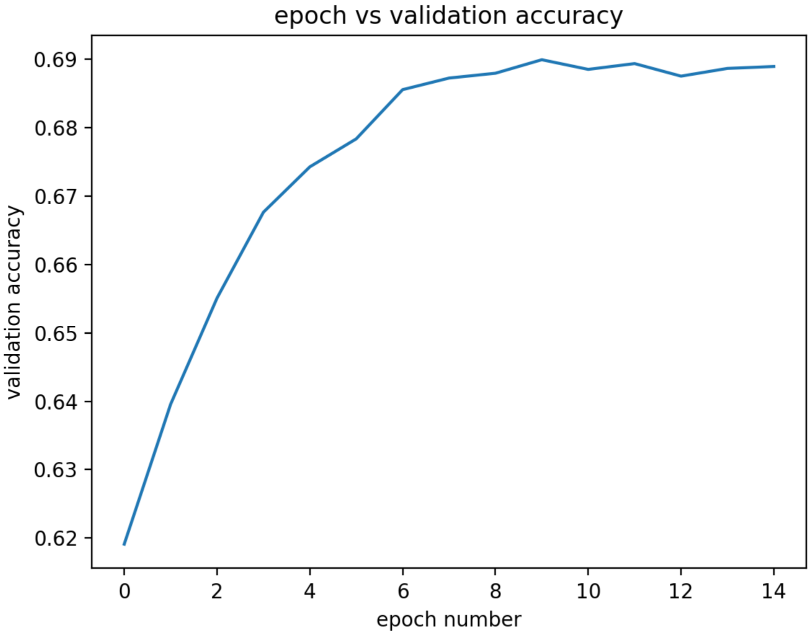
Question 3d (Neural Networks)

We chose $\boxed{k^* = 10}$.

The graph of epoch vs training loss is:



The graph of epoch vs validation loss is:



Below is the training process

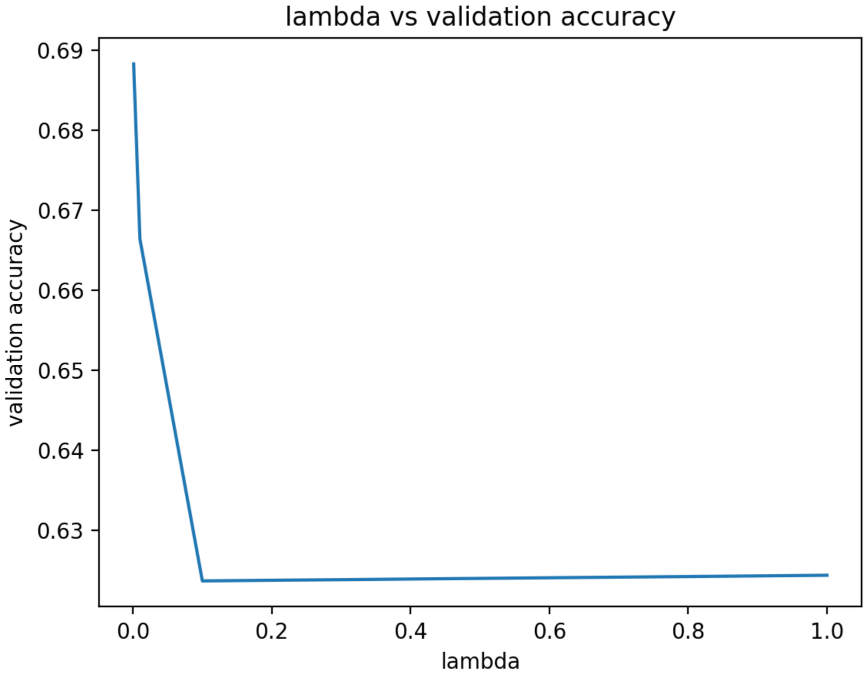
```
Epoch: 0   Training Cost: 13499.529709 Valid Acc: 0.6191081004798193
Epoch: 1   Training Cost: 12297.436962 Valid Acc: 0.6395709850409258
Epoch: 2   Training Cost: 11572.161254 Valid Acc: 0.6550945526390065
Epoch: 3   Training Cost: 10948.248247 Valid Acc: 0.6676545300592718
Epoch: 4   Training Cost: 10443.843648 Valid Acc: 0.6742873271239063
Epoch: 5   Training Cost: 10038.862250 Valid Acc: 0.6783799040361276
Epoch: 6   Training Cost: 9706.154937 Valid Acc: 0.6855771944679651
Epoch: 7   Training Cost: 9417.888905 Valid Acc: 0.6872706745695738
Epoch: 8   Training Cost: 9170.807286 Valid Acc: 0.6879762912785775
Epoch: 9   Training Cost: 8949.011406 Valid Acc: 0.6899520180637877
Epoch: 10  Training Cost: 8744.179799 Valid Acc: 0.6885407846457804
Epoch: 11  Training Cost: 8560.910469 Valid Acc: 0.6893875246965848
Epoch: 12  Training Cost: 8395.020434 Valid Acc: 0.6875529212531752
Epoch: 13  Training Cost: 8250.481424 Valid Acc: 0.6886819079875811
Epoch: 14  Training Cost: 8096.418642 Valid Acc: 0.6889641546711827
```

$\boxed{\text{The final } \textbf{test accuracy} \text{ is } 0.6876}$.

Question 3e (Neural Networks)

We chose learning rate = 0.1, number of epoch = 15, $k = 10$
The graph of lambda vs test accuracy is:

Parameter Name	Value
k	10
Learning Rate	0.1
# epoch	15



Below is a table of lambda and validation accuracy:

λ	Validation Accuracy
0.001	0.6883
0.01	0.6664
0.1	0.6236
1	0.6243

We chose $\lambda = 0.001$. The final **validation accuracy** and **test accuracy** are as below:

λ	validation accuracy	test accuracy
0.001	0.6883	0.6895

Comparing to Q3 ii (d), the test accuracy does not change a lot, and they differ by about 0.002. We think this is because the model does not overfit, and hence adding a regularizer (that essentially does weight decay to reduce model complexity) did not help much. Also, we chose the model that has the highest validation accuracy, that's the reason why it is unlikely to overfit.

Question 4

Base Model Selection: The three base models that we chose are

(1) The kNN model from Q1

(2) The IRT model from Q2

(3) The autoencoder from Q3

Final Validation and Test Accuracy:

Validation Accuracy	Test Accuracy
0.7062	0.7039

Bootstrapping Method: In our bootstrapping process, we bootstrapped by **entries**, that is, we randomly sampled (student, question) pairs **with replacement** from the training data dictionary to form new datasets, each of size the same as original training data.

Ensembling Process: For each base model, we used a different bootstrapped dataset. We then separately trained each of our base model with the hyperparameters as discovered from previous questions, and made predictions using each base model. We then took the weighted average of each prediction in order to form our final prediction. The specific weights we assign are results of tracking validation accuracy, which are

Model	kNN	IRT	Auto-Encoder
Weights	1/4	1/2	1/4

The results we obtained from the ensemble is similar to the best of the three base models (i.e. IRT). We believe that this happens because the IRT model consistently performs better than the other two (which is why we returns out weight it more), hence there are limited performance boost from the other two models giving to the IRT model.

Part (B)

1. Formal Description of Methods

In this section, we will discuss how do we formulate a model imporved from part A, step by step. In section 1.1, we will talk about how and why do we change the auto-encoder (NN) from part A (user as entity) to be question as entity. In section 1.2 and 1.3, we discuss about how to enrich the latent information of a question. More specifically, section 1.2 talks about utilize the metadata, while the section 1.3 attempts to incorporate parameters from a pre-trained IRT model. Next, the section 1.4 discuss the optimization method – adopting mini-batch gradient decent instead of stochastic gradient decent. Lastly, in section 1.5, we discuss how do we change the method of replacing originally missing data (NaN) to a more reasonable numbers (in our belief).

1.1. AE with Question as Entity

For this part of our project, we decided to modify the **auto-encoder** algorithm from Part A. The first modification we made was that we used questions as entities (instead of users). Our auto-encoder adopted the 3-layer architecture from Part A. The first layer is the input vector, representing the correctness of each user on one specific question; the second layer is the latent code of the question, and the third layer is the reconstruction.

Formally, let N_{users} denotes the number of users, and $\mathbf{v}_j \in \mathbb{R}^{N_{\text{users}}}$ denotes a question j, and k denotes the latent dimension of the latent code for vector $\mathbf{v}_j$. One forward pass of the entire neural network for this question j would be

$$\begin{aligned}\mathbf{z}_{\mathbf{v}_j} &= \mathrm{Enc}\left(\mathbf{v}_j\right) := \sigma\left(\mathbf{W}^{(1)}\mathbf{v}_j + \mathbf{b}^{(1)}\right),\\ \widetilde{\mathbf{v}_j} &= \mathrm{Dec}\left(\mathbf{z}_{\mathbf{v}_j}\right) := \sigma\left(\mathbf{W}^{(2)}\mathbf{z}_{\mathbf{v}_j} + \mathbf{b}^{(2)}\right),\end{aligned}$$

where

- $\mathrm{Enc} : \mathbb{R}^{N_{\text{users}}} \to \mathbb{R}^k$ is the encoder,
- $\mathrm{Dec} : \mathbb{R}^k \to \mathbb{R}^{N_{\text{users}}}$ is the decoder,
- $\mathbf{z}_{\mathbf{v}_j} \in \mathbb{R}^k$ is the learned latent representation of vector $\mathbf{v}_j$,
- $\sigma(.)$ denotes the sigmoid activation function,
- $\left\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}\right\}$ are the learnable parameters of the neural network (weights and biases).

Since our goal is to reconstruct the input vector $\mathbf{v}_j$, the loss is computed as

$$||\mathbf{v}_j - \widetilde{\mathbf{v}_j}||_2^2$$

for each reconstruction $\widetilde{\mathbf{v}_j}$ of $\mathbf{v}_j$.

Note for computing the loss: Same as the goal in part A, our task is to complete the originally missing entries of the vector $\mathbf{v}_j$ while reconstructing the observed entries (not missing-valued) of the vector, so the reconstruction loss between $\mathbf{v}_j$ and $\widetilde{\mathbf{v}_j}$ is computed only on those observed entries of $\mathbf{v}_j$. I.e., the reconstruction is supervised on the non-missing entries of the question vector.

We decided to have question as entity instead of user (as in Part A) to make a distinction from what we did in Part A. This model will serve as a baseline for us to implement other ideas upon, and to be compared against. After implementation, we found that the performance of this modified model achieves similar accuracy as the auto-encoder in Part A. Though the performances are similar, we believe that this model (having question as entity) is easier to extend. One obvious example being the metadata for question is more *complete* than the metadata for students, i.e. there are much less missing entries in question metadata compare to its student metadata counterparts.

1.2. Meta Data Encoder Jointly Trained

In addition to the training data, we are given with a series of metadata. More specifically, in this section, we discuss how we incorporate the question metadata into our neural network pipeline. The question metadata – originally given as a few class labels representing which subjects each question belongs – is theoretically beneficial to our question-based autoencoder as the model could treat questions with similar subject labels in a similar manner, potentially. To use this metadata, we first need to pre-process it. We initially convert each question's subject labels (as discrete scalars) into a vector of # of subjects labels dimension, filled by 1s and 0s only, where the i-th entry being 1 means that this particular question covers the i-th topic in original raw metadata (representing each question's metadata this way gives us a matrix). However, since each vector is so sparse – a vector of with around 300 dimensions but filled by 1s in only 4-5 entries, we believe that we can reduce its dimension without great loss of information, and also make it easier for the network to pick up the true pattern of this metadata. More specifically, our approach is to **jointly train two encoders**, one of which being our ordinary auto-encoder from section 1.1 that encodes $\mathbf{v}_j$, and the other one encodes the pre-processed question metadata, denoted as $\mathbf{m}_j$ into a lower dimensional latent space, for each question j. We then concatenate the two latent codes $\mathbf{v}_j$ and $\mathbf{m}_j$ to form the latent code of question j, and let the decoder decode it into our reconstruction and prediction.

Formally, let N_{users} denotes the number of users, N_{subjects} denotes the number of subjects. Let $\mathbf{v}_j \in N_{\text{users}}$ denotes answer correctness from all users for a question j, and let $\mathbf{m}_j \in N_{\text{subjects}}$ denotes the subjects coverage for a question j (as described above, a vector of 1s and 0s). Let k and k_m denote the latent dimension of the latent code for vector $\mathbf{v}_j$ and metadata $\mathbf{m}_j$, respectively. One forward pass of the entire neural network would be

$$\begin{aligned}
\mathbf{z}_{\mathbf{v}_j} &= \text{Enc}(\mathbf{v}_j) := \sigma\left(\mathbf{W}_{\mathbf{v}}^{(1)}\mathbf{v}_j + \mathbf{b}_{\mathbf{v}}^{(1)}\right), \\
\mathbf{z}_{\mathbf{m}_j} &= \text{EncMeta}(\mathbf{m}_j) := \sigma\left(\mathbf{W}_{\mathbf{m}}^{(1)}\mathbf{m}_j + \mathbf{b}_{\mathbf{m}}^{(1)}\right), \\
\mathbf{z}_j &= \text{Concat}\left(\mathbf{z}_{\mathbf{v}_j}, \mathbf{z}_{\mathbf{m}_j}\right) \\
\widetilde{\mathbf{v}_j} &= \text{Dec}(\mathbf{z}_j) := \sigma\left(\mathbf{W}^{(2)}\mathbf{z}_j + \mathbf{b}^{(2)}\right),
\end{aligned}$$

where

- $\text{Enc} : \mathbb{R}^{N_{\text{users}}} \to \mathbb{R}^k$ is the encoder as before
- $\text{EncMeta} : \mathbb{R}^{N_{\text{subjects}}} \to \mathbb{R}^{k_m}$ is the encoder for question metadata
- Concat(.) denotes the concatenation of all input as a vector output
- $\text{Dec} : \mathbb{R}^{k+k_m} \to \mathbb{R}^{N_{\text{subjects}}}$ is the decoder
- $\mathbf{z}_{\mathbf{v}_j} \in \mathbb{R}^k$ is the learned latent representation of vector $\mathbf{v}_j$
- $\mathbf{z}_{\mathbf{m}_j} \in \mathbb{R}^{k_m}$ is the learned latent representation of vector $\mathbf{m}_j$
- $\mathbf{z}_j$ is the latent code of the question j, concatenated from $\mathbf{z}_{\mathbf{v}_j}$ and $\mathbf{z}_{\mathbf{m}_j}$
- $\sigma(\cdot)$ denotes the sigmoid activation function
- $\left\{\mathbf{W}_{\mathbf{v}}^{(1)}, \mathbf{b}_{\mathbf{v}}^{(1)}, \mathbf{W}_{\mathbf{m}}^{(1)}, \mathbf{b}_{\mathbf{m}}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}\right\}$ are the learnable parameters of the neural network (i.e. weights and biases)

The reason why we choose to train the model this way is that we believe the metadata encoder would extract significant latent information about a specific question, and thus can help the reconstructions of the data and more easily distinguishes between different questions (compare to only have vector $\mathbf{v}_j \in \mathbb{R}^{N_{\text{users}}}$) for a better predictions at the originally missing entries. Instead of pre-training an auto-encoder for the metadata entirely on its own, we believe that the "jointly train" allows two encoder to be optimized collaboratively by adjusting their parameters that could potentially be correlated mutually, for both a better latent representation and final reconstruction & predictions, during the training process (gradient decent). Injecting this information may give us a latent layer of higher quality, thus optimizing our model.

1.3. IRT Injection into Latent

With a pre-trained IRT model, we get a trained vector $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_{N_{\text{question}}})$, and each entry β_j is associated to each question j, representing the estimated difficulty of the question j. We deem it as a hidden variable (or latent information) of the question j. With the belief that this parameter is meaningful as a latent information for the question, we append it to the latent code compressed from the encoder in previous part, and treat this as an extra dimension of the question latent space, forcing the decoder decode this appended latent code.

Formally, the minor modification from the formulation in section 1.2, is to represent the latent of a question j as a vector $\mathbf{z}_j$ of dimension $k + k_m + 1$, as a result of concatenating $\mathbf{z}_{\mathbf{v}_j}$, $\mathbf{z}_{m_j}$, and scalar β_j, written as

$$\begin{aligned}
\mathbf{z}_{\mathbf{v}_j} &= \mathrm{Enc}(\mathbf{v}_j) := \sigma\left(\mathbf{W}_{\mathbf{v}}^{(1)}\mathbf{v}_j + \mathbf{b}_{\mathbf{v}}^{(1)}\right), \\
\mathbf{z}_{\mathbf{m}_j} &= \mathrm{EncMeta}(\mathbf{m}_j) := \sigma\left(\mathbf{W}_{\mathbf{m}}^{(1)}\mathbf{m}_j + \mathbf{b}_{\mathbf{m}}^{(1)}\right), \\
\mathbf{z}_j &= \mathrm{Concat}\left(\mathbf{z}_{\mathbf{v}_j}, \mathbf{z}_{\mathbf{m}_j}, \boxed{\beta_j}\right) \\
\widetilde{\mathbf{v}_j} &= \mathrm{Dec}(\mathbf{z}_j) := \sigma\left(\mathbf{W}^{(2)}\mathbf{z}_j + \mathbf{b}^{(2)}\right),
\end{aligned}$$

where all the notations are consistent with previous formulation.

The reason why we decided to inject IRT parameters into this model is because by our Part A accuracies, the IRT model performs the highest, this means that the IRT parameters are valuable to some extent. Moreover, since the IRT parameter that we used (i.e. β), reflects the "difficulty" of a certain question, injecting this parameter to the model gives our model the estimated difficulty of a certain question, which in turn enables the model to make better predictions based on this extra piece of information.

1.4. MiniBatch Gradient Descent

In Part A, the neural network uses *stochastic gradient descent* for optimization. We decided to shift to *minibatch gradient descent* for **less noisy steps** during gradient decent, as well as exploiting parallelism for a faster training time. We believe this change of optimization dynamic would improve the accuracy by a margin, as each step is more representative of the dataset and contains less noise.

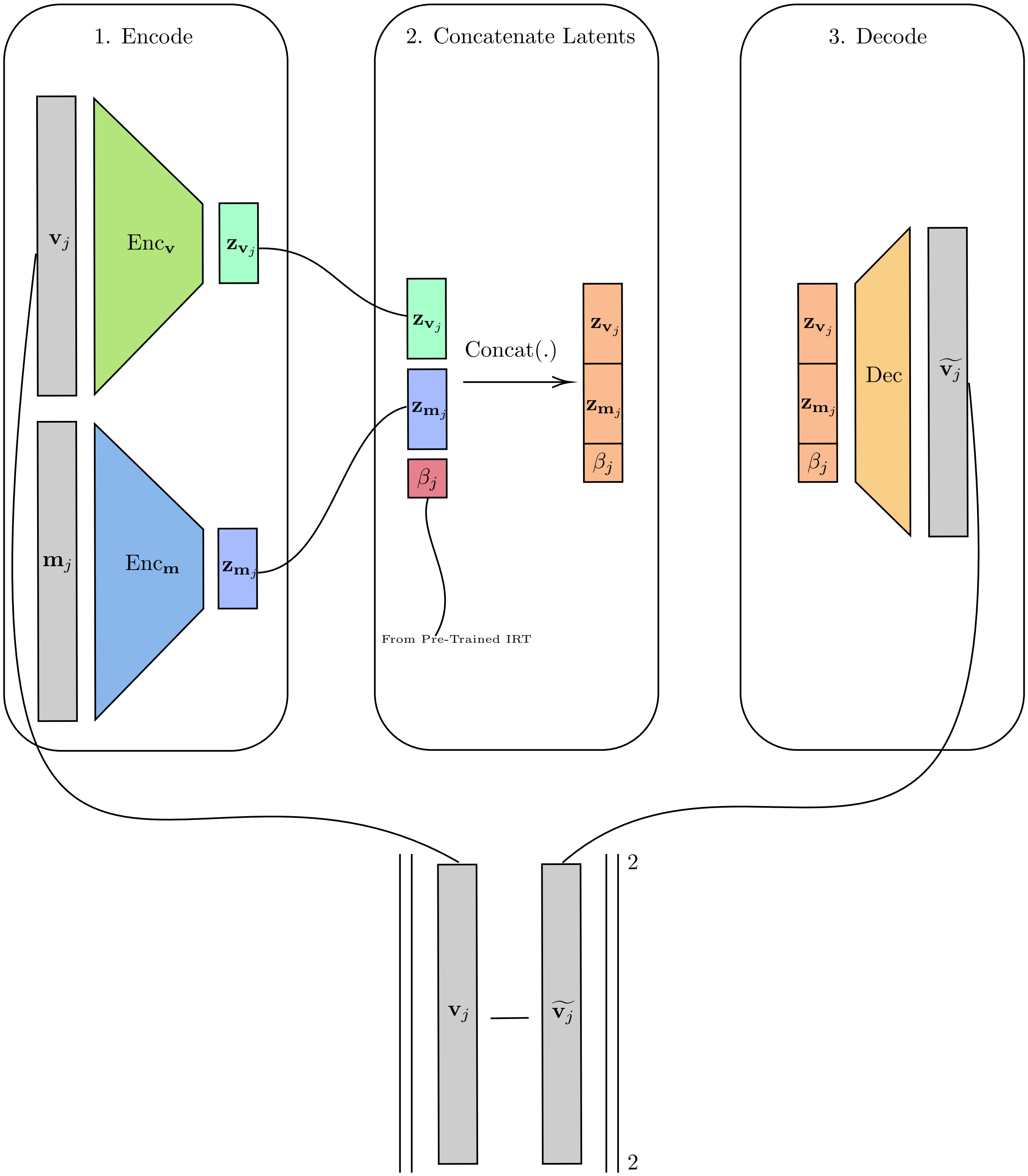
1.5. Replacing NaN with MLE of Bernoulli Trial

In Part A, before we feed the vector $\mathbf{v}_j$ through the neural network, we replaced all of its missing entries (NaN-values) with 0s. However, we believe that replacing NaN with 0s is not very reasonable, since this would mislead the **encoder** of vector $\mathbf{v}_j$ treating incorrect-answers and no-answers exactly the same way. Although replacing NaN's with 0s are not reasonable, we still want to assign **the same value** to each question's NaN entries since this prevents us from giving different biases to different users – if such bias is not accurate, then the prediction result could be affected in a negative way. Among all possible values we could pick, the *maximum likelihood estimate* of the success parameter of Bernoulli trial is a natural choice, because it represents the empirical correct-answer rate of the question. Not knowing whether a user would get a certain question correct, we statistically treat this as "tossing an unfair coin", and what we know from the observed data is how many times have we landed in "heads" and "tails" using this "coin". In this metaphor landing in "heads" means the student answers correctly, and landing in "tails" means the student answers incorrectly. In pseudocode:

```
for question_vector in train_matrix do
    empirical_mean <- mean of question_vector ignoring NaN values
    for entry in question_vector do
        if entry is NaN then entry <- empirical_mean
```

The reason why we think that the performance would be better than filling NaNs with 0s is that (1) it distinguishes missing-value from the value 0 which stands for a wrong answer, (2) the maximum likelihood estimate is more reflective of whether the question *could be* correctly answered.

2. Figure


1. Encode
$\mathbf{v}_j$
$\mathrm{Enc}_{\mathbf{v}}$
$\mathbf{z}_{\mathbf{v}_j}$
$\mathbf{m}_j$
$\mathrm{Enc}_{\mathbf{m}}$
$\mathbf{z}_{\mathbf{m}_j}$
2. Concatenate Latents
$\mathbf{z}_{\mathbf{v}_j}$
$\mathbf{z}_{\mathbf{m}_j}$
β_j
Concat(.)
From Pre-Trained IRT
3. Decode
Dec
$\widetilde{\mathbf{v}_j}$
$\| \mathbf{v}_j - \widetilde{\mathbf{v}_j} \|_2^2$

This figure shows the architecture of our auto-encoder.

- In the encoding part, the two encoders takes in the question vector $\mathbf{v}_j$ and the question metadata vector $\mathbf{m}_j$ and encode them to $\mathbf{z}_{\mathbf{v}_j}$ and $\mathbf{z}_{\mathbf{m}_j}$, respectively.
- After encoding, we concatenate those latents, as well as the β parameter from our pre-trained IRT model.

- We then decode the concatenated latent to reconstruct the original question vector.
- We finally compute the squared L_2 loss between the original question vector and reconstructed question vector.

3. Comparision/Demonstration

3.1. Specification for different models used for experiement

Model	Model Specification
(1) NN – User as Entity (Part A)	This model is exactly the same as Part A - Question 3.
(2) IRT (Part A)	This model is exactly the same as Part A - Question 2.
(3) NN – Question as Entity (Baseline)	**This model our baseline model** – the one discussed in section 1.1, it has question as entity rather than user.
(4) NN + Metadata Injection	This model is the one discussed in section 1.2, having a jointly trained auto-encoder for raw metadata, on top of the baseline model (3).
(5) NN + IRT Injection	This model is the one adopts the idea of appended latent with β_j scalar from pre-trained IRT model. It only adds a scalar β_j to latent $\mathbf{z}_{\mathbf{v}_j}$ to form the latent of a question j. That is, it is **not exactly the same model as formally shown in section 1.3**, but only an add-on of "beta injection into latent" on top of the baseline model (3). We can still verify our idea in section from 1.3 by comparing this model with our baseline model though.
(6) NN + MiniBatch Gradient Descent	This model corresponds to method of section 1.4., just changes stochastic gradient decent from the baseline model (3) to mini-batched gradient decent.
(7) NN + MLE	This model corresponds to method of section 1.5., replacing the NaN missing entry with MLE of bernuolli trail rather than 0, and then train with our baseline model (3).
(8) NN + Metadata + IRT + MBGD + MLE	This model uses all of our idea from section 1 — based on formulation of section 1.3, we also use mini-batched gradient decent (section 1.4) and replacing missing value with MLE (section 1.5).

3.2. Comparision/Demonstration

Model	Hyperparameters	Validation Accuracy	Test Accuracy

(1) NN – User as Entity (Part A)	$k = 10$
	LR = 0.1
	Epochs = 15
		0.6890	0.6876

(2) IRT (Part A)	LR = 0.01
	Epochs = 25
		0.7065	0.7053

(3) NN – Question as Entity (Baseline)	$k = 20$
	LR = 0.01
	Epochs = 15
		0.6952	0.6881

(4) NN + Metadata Injection	$k = 30$
	$k_m = 3$
	LR = 0.01
	Epochs = 15
		0.6988	0.6901

(5) NN + IRT Injection	$k = 20$
	LR = 0.01
	Epochs = 15
		0.7130	0.7082

(6) NN + MiniBatch Gradient Descent	$k = 10$
	Batch size = 8
	LR = 0.001
	Epochs = 20
		0.7028	0.6890

(7) NN + MLE	$k = 10$
	LR = 0.01
	Epochs = 20
		0.7019	0.6943

(8) NN + Metadata + IRT + MBGD + MLE	$k = 12$ $k_m = 5$ Batch size = 10 LR = 0.001 Epochs = 15	0.7180 (highest)	0.7090 (highest)

3.3. Result Analysis & Hypothesis in Section 1 Got Verified:

In this section, we complement some analysis of our experiment results and explain why we have verified our hypothesis on our proposed method from section 1 would work well.

- For question (1) and (3) illustrate our claim in section 1 that using NN with user or question as entity achieve similar accuracy, and we go with model (3) – question as entity – as our baseline model since it is easier to extend in our perspective.

- For Metadata Injection (the jointly trained autoencoder idea from section 1.2), our hypothesis was proved to work by experiment with model (4), seeing that this method yielded both higher validation and test accuracy than our baseline model (3), as shown in the table. This method gives our model **additional relevant information** about a question entity, consequently improving our accuracy score.

- For IRT injection (the β appended latent code idea from section 1.3), our hypothesis was proved to work by experiment with model (5), seeing that this method yielded both higher validation and test accuracy than two baseline models – IRT (2) and NN with User as Entity (3), as shown in the table.

  Firstly, model (5) performs better than baseline model (3) indicates that our baseline auto-encoder has not learned the representation of question difficulty as well as the β scalar from trained IRT.

  Secondly, model (5) performs better than baseline model (2) indicates that the latent layers of an auto-encoder contains more insights than just the parameter β from IRT for a question.

This method gives our model **additional relevant information** to work with, consequently improving our accuracy score.

- For MiniBatch Gradient Descent (1.4), a small increase of accuracy in model (6) from the baseline model indicates that this approach is working. This method is **improving the optimization** on our baseline model since it improves our approach of optimization by less noisy steps than SGD (i.e. from SGD to MBGD + Adam Optimizer)
- For MLE (1.5), our hypothesis was proved to work by experiment, seeing that model (7) yielded a higher accuracy than our baseline. This method is performing **additional data preprocessing** on our baseline model, it to some extends proves that it helps distinguishes the 0s from NaN's, and gives us a better estimate on the unseen entries (i.e. assigning the MLE instead of assigning an arbitrary value of 0).
- Finally, putting all of ideas from section 1 altogether in model (8), we achieve the best accuracy of all, in both validation and test dataset.

4. Limitations

From our metadata jointly trained encoder from section 1.2, the increased computational cost may inhibit us from finding a good set of hyperparameters, which could impact the model performance

- The settings in which we'd expect this approach to perform poorly is that if we don't have an abundant amount of computational resources. Using this approach, since we are jointly training two autoencoders, it is empirically a lot slower than without metadata injection, therefore it is costly to tune hyperparameters. With less tuning of hyperparamters our model would not perform as good as it potentially could.
- This limitation exists because both theoretically and empirically, jointly training another autoencoder is more computationally expensive, and it becomes less feasible for us to tune the hyperparameters. Additionally, jointly training another autoencoder introduces another hyperparameter, which is the latent dimension of the newly added autoencoder. Combining the facts, it is both more costly to perform one iteration of hyperparameter tuning, AND there are more possibilities of hyperparameters to tune from.
- One way to address this limitation is to increase computational power, so that we could better tune the hyperparameters to unleash the full potential of this approach. One possible extension is to **pretrain** the latent representations for metadata. Though pre-training could be as costly as jointly trained, it is only done *once*.

From our IRT injection into latent space in section 1.3, an apparent limitation is that how good our model is dependent on the quality of the IRT parameters.

- The settings in which we'd expect this approach to perform poorly is that if we had a poorly pretrained IRT model, in which case the IRT parameters would not be useful. Since the latent space encodes important information, injecting a useless parameter into it would bring confusion to the model.
- This limitation exists because poor IRT pre-trained parameters could misleads the auto-encoder's reconstruction and prediction.
- One way to address this limitation is to pretrain a good IRT model in the first place. To take one step beyond, we could adopt a more complex version of IRT (e.g. the three-parameter logistic model) for a greater hypothesis space, which could potentially give higher-quality IRT parameters.

From our switch to the usage of Minibatch Gradient Descent, introducing another hyperparameter (i.e. batch size) may increase the computational cost of finding the best set of hyperparameters.

- The settings in which we'd expect this approach to perform poorly is that if we don't have an abundant amount of computational resources. By introducing another hyperparameter, the computational cost of grid searching would be multiplied by the number of candidate values of batch sizes.
- This limitation exists because it is clear that having one more hyperparameter means that there are more sets of hyperparameters to try, which imposes a computational cost
- One way to address this limitation is to increase computational power, so that we could better tune the hyperparameters to unleast the full potential of this approach.

From our idea of replacing NaN with MLE of Bernoulli Trial, it is an open question how to fill those missing values. What we did was filling **all missing entries** with the MLE based on the proportion of users correctly answering **a specific question**. However, we acknowledge that we could have filled the different missing entry of one question vector differently, by incorporating the associated user's information.

- An example of a setting in which we'd expect our approach to perform poorly is when the question we are estimating is an easy calculus question which is correctly answered by all university students, however, if we were to predict how well an elementary school student would answer this question, it is more likely that they would not have answered it correctly.

- This limitation exists because a model could only take account in certain aspects of this problem. If we were to use another model, it might have overcome the situation as described above, but it could fail on other situations where our current model performs well. Let's say that we chose the MLE based on the proportion of questions correctly answered by a specific user, then there are other scenarios where the approach fails (i.e. fixing the student to be a high-achieving elementary school student, they would not be able to correctly answer questions in the field of differential geometry).
- This problem is an open problem. Optimizing the choice of what to use to replace the NaN value could be achieved by some advanced statistic techniques, where we rigorously analyse the distribution of the data in advance.