MLA Midterm Project

Tutorial Class 06 - Group 3

October 2024

Part A: Machine Learning Models

A.1: k-Nearest Neighbor (kNN)

For the full code reference, please take a look at the `knn.py` file in the `a1_k-Nearest Neighbor` folder.

The code implements the k-Nearest Neighbors (kNN) algorithm for user-based collaborative filtering in the `knn_impute_by_user` function. Here's how it works:

a) Model explanation

The `knn_impute_by_user` function uses **KNNImputer** from **scikit-learn** to fill in missing values based on student similarity (Euclidean distance). The sparse matrix (which contains students' responses to questions) is passed to the imputer, and it fills in missing entries by finding the k-nearest students for each missing value. After filling the missing values, the function evaluates the accuracy of the predictions using a validation dataset through the `sparse_matrix_evaluate` function, which compares the imputed values against real responses.

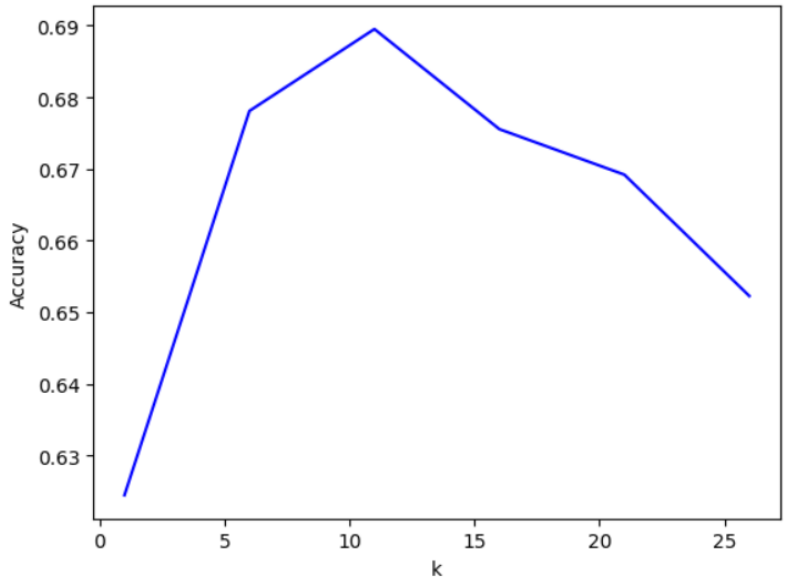
Meanwhile, the function `user_based_report_knn` takes a list of k values [1, 6, 11, 16, 21, 26] and evaluates the accuracy for each one using the validation dataset. It generates a plot with k values on the x-axis and the corresponding accuracy values on the y-axis, allowing us to visualize how the accuracy changes as k increases. This helps to choose the optimal k (the one with the highest accuracy). The code dynamically selects the best k based on the maximum accuracy and then reports the accuracy on the test dataset.

Regarding item-based filtering, it compares questions instead of comparing students. The idea is that a student's performance on a particular question can be predicted by looking at how they performed on similar questions. The `knn_impute_by_item` function transposes the sparse matrix so that rows represent questions (items) instead of students. Then, it applies kNN using the KNNImputer to find similar questions and predict missing values. After imputation, the matrix is transposed back to its original form to evaluate accuracy

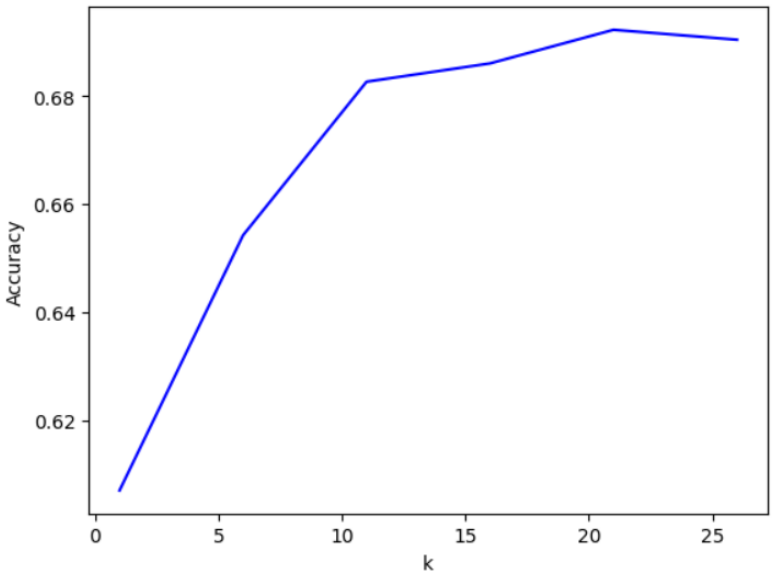
based on students.

Similar to user-based filtering, `item_based_report_knn` computes accuracy for different values of k, plots the results, and selects the best k for testing on the test dataset.

The test accuracy value for both user-based and item-based filtering are shown in the figures below:



Test value accuracy of user-based filtering, with the highest value of 0.6842 at $k = 11$



Test value accuracy of item-based filtering, with the highest value of 0.6816 at $k = 21$

b) Comparisons between the two filtering methods

The user-based approach slightly outperforms the item-based approach with a test accuracy of 0.6842 compared to 0.6816. While both methods have similar results, user-based filtering is marginally better in this case. There are 2 reasons leading to the user-based filter being better:

- **Student Similarity**: The user-based method might be more effective in this dataset because students who tend to answer similarly likely share similar learning patterns or knowledge levels, leading to more accurate predictions based on their past responses.
- **Fewer Missing Data**: It's possible that there are more overlapping responses among students than among questions. In this case, user-based filtering could benefit from a larger pool of similar neighbors to make accurate predictions.

However, the item-based filter is still close enough compared to its user-based counterpart, mainly because of these 2 factors:

- **Question Similarity**: The item-based filtering method still performed well, which could indicate that questions in the dataset share structural or topical similarities. For example, if certain types of questions (e.g., similar difficulty or format) lead to consistent performance patterns, item-based filtering can make effective predictions.

- **Chosen k**: The chosen k for item-based filtering was 21, which is relatively high. This suggests that finding a larger group of similar questions was necessary to make good predictions, possibly due to less direct similarity between smaller groups of questions.

To sum up, while the difference in accuracy between user-based and item-based filtering is small, user-based filtering performed slightly better. This might be because students' behavior and response patterns are more predictable when using their peers' answers as references. However, item-based filtering is still a viable approach, especially if question similarities play a major role in performance. In practice, which method works better may depend on the nature of the dataset (student consistency vs. question structure).

A.2: Item Response Theory (IRT)

For this model, we will divide it into separate parts, so that the process is understood more clearly. For the full code reference, please take a look at the `item_response.py` file in the `a2_Item Response` folder.

a) For the first step of the model implementation, we assume that the probability of a correct answer is given by:

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

where θ_i represents the ability of student i, and β_j represents the difficulty of question j.

This formula has been shown in the original problem description. As a result, proof for this is not needed.

b) Completing the `neg_log_likelihood` Function

The negative log-likelihood function is completed to calculate the sum of log-probabilities over the dataset. The model calculates the likelihood of students answering questions correctly based on the current θ and β parameters.
By adding the log-likelihoods across all student-question pairs, the function returns the negative value, which is used as the objective to minimize during training.

c) Introducing Parameter Updates (Gradient Descent)

One of the critical missing components was the ability to update the model's parameters using gradient descent. The `update_theta_beta` function was introduced to:

- Compute the gradients with respect to student abilities (θ) and question difficulties (β).
- Update these parameters iteratively using a learning rate (a hyper- parameter controlling the step size) to minimize the negative log-likelihood.

This function ensures that both parameters are adjusted after each iteration to improve the model's predictions.

d) Adding the Model Training Function
The core of the integration was the development of the `irt` function, which handles the model training process. This function:

- Initializes the θ and β vectors.
- Iteratively updates these parameters using the gradient descent method.
- Tracks the negative log-likelihood and accuracy at each iteration to monitor the model's convergence.

The function is designed to terminate after a specified number of iterations, producing a trained model that can then be evaluated on validation and test datasets.

e) Evaluation and Performance Metrics
To assess the accuracy of the trained model, an `evaluate` function was added. This function:

- Computes the predicted probabilities for each student-question pair.
- Compares these predictions to the actual responses (correct or incorrect) and calculates the accuracy.

The accuracy metric allows us to validate the effectiveness of the model on unseen data and ensures that the learned parameters generalize well.

f) Visualization
After training, the model's performance is visualized by plotting:

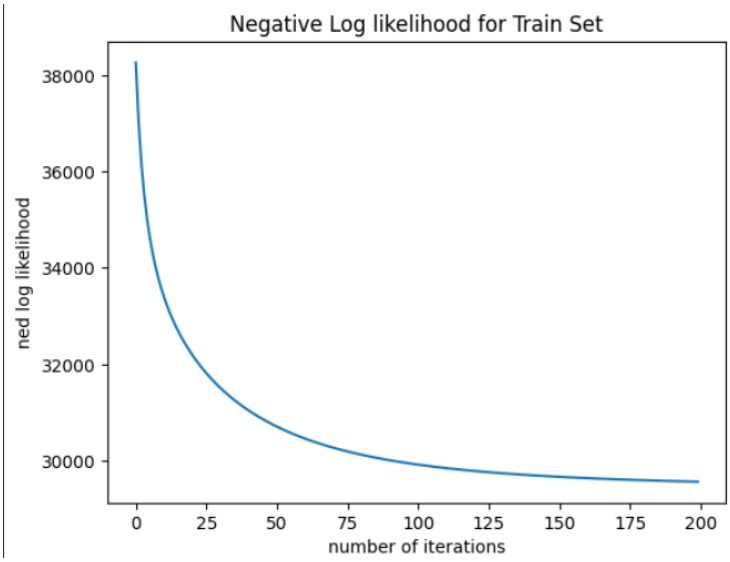
- The negative log-likelihood over the iterations to observe the convergence behavior.
- The probability of answering questions correctly as a function of student ability for a few selected questions, giving a deeper insight into the relationship between student ability and question difficulty.

g) Hyper-parameters selected for IRT and output

```
hyperparameters:
 num_iterations = 200, learning rate = 0.00255
```

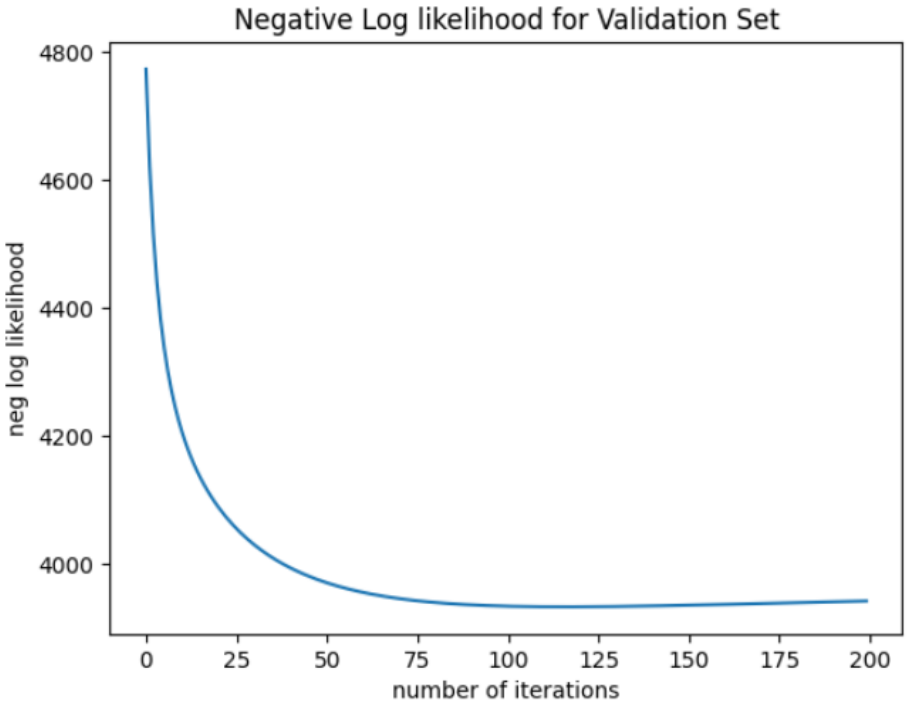
(1)

Hyper-parameters selected for IRT (2)



(3)

Negative log-likelihood for each iteration of train set (4)



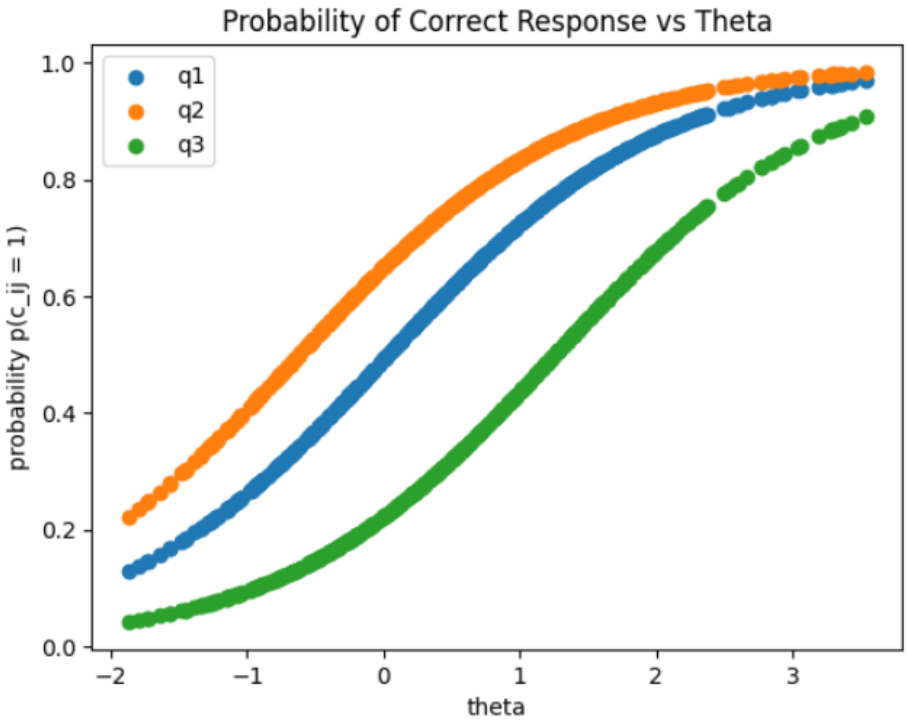
(5)

Negative log-likelihood for each iteration of validation set

```
val accuracy:
0.70575783234547
test accuracy:
0.7067456957380751
```

(6)

Final validation and test accuracy



Plot for the probability of getting correct response vs theta for 3 questions

The graph illustrates the relationship between theta and the probability of correctly answering three questions (q1, q2, and q3). The curves, shaped like a logistic function, demonstrate that as theta increases, so does the likelihood of a correct response. However, the relative positions of the curves indicate varying levels of difficulty among the questions, with q1 being the easiest and q3 the most challenging. Individual differences in performance are also evident, suggesting factors beyond theta, such as personal characteristics or test-taking strategies, influence the probability of success.

A.3: Ensemble Learning

To see how this is implemented using Python code, please check out the `ensemble.py` file in `part_a` folder.
The process for this part is as follows:
Step 1: Base Model Selection The following three base models were selected for this ensemble method:

- k-Nearest Neighbor (KNN)
- Item Response Theory (IRT)

- Neural Network

Each model was trained independently on the training dataset, yielding the following baseline accuracy values:

- KNN: 0.6842
- IRT: 0.7067
- Neural Network: 0.6890

Step 2: Bootstrap Aggregating (Bagging)
In this step, bootstrap aggregating was applied to each model. This process involved creating multiple versions of the training set by sampling with replacement. For the implementation, three bootstrapped datasets were generated for each base model, which were then used to train the respective models.

Step 3: Prediction
For each test sample, predictions were made on whether a student would answer a question correctly using each of the three base models. The final prediction was computed by averaging the predictions (or taking a majority vote) from KNN, IRT, and Neural Network.

Step 4: Performance Evaluation
The performance of the ensemble model was evaluated through the following accuracy measures on the validation and test datasets:

- KNN Validation Accuracy: 0.6511
- IRT Validation Accuracy: 0.7005
- Neural Network Validation Accuracy: 0.6818

The final results from the ensemble method were as follows:

- Ensemble Validation Accuracy: 0.6898
- Ensemble Test Accuracy: 0.6946

These results indicate that the ensemble method improved the validation accuracy slightly compared to the KNN and Neural Network, though not surpassing the IRT's accuracy. However, the ensemble accuracy is competitive and demonstrates a general enhancement in prediction stability.

Step 5: Discussion

a) Advantages of Using Ensemble Methods

- **Reduction of Overfitting**: Bagging helps mitigate overfitting by training individual models on varied samples of the training data.

- **Improved Prediction Stability**: The averaging of predictions leads to more consistent results, particularly in the presence of noise.
- **Diversity in Predictions**: Each model may learn different patterns, contributing to a more comprehensive understanding of the data.

b) Limitations of the Ensemble Model

- **Increased Complexity**: The ensemble method increases computational complexity and requires more resources for training and prediction.
- **Diminishing Returns**: If the base models are too similar, the benefits of bagging may be limited, resulting in minimal improvements.
- **Model Interpretability**: The aggregation of multiple models may reduce the overall interpretability of predictions, making it harder to identify the influence of individual models.

A.4: Matrix Factorization or Neural Networks

For this part, we have decided to go for a neural network model. Hence, the file `matrix_factorization.py` is unchanged and only the file `neural_network.py` has been edited, both of which can be accessed via the `part_a` folder.

For the first step, an AutoEncoder is implemented, the code for this component is shown below:

```python
class AutoEncoder(nn.Module):  2 usages
    def __init__(self, num_question, k=100):
        """ Initialize a class AutoEncoder.

        :param num_question: int
        :param k: int
        """
        super(AutoEncoder, self).__init__()

        # Define linear functions.
        self.g = nn.Linear(num_question, k)
        self.h = nn.Linear(k, num_question)
        self.k = k

        self.encoder = torch.nn.Sequential(
            self.g,
            torch.nn.Sigmoid()
        )

        self.decoder = torch.nn.Sequential(
            self.h,
            torch.nn.Sigmoid()
        )
```

Code for the AutoEncoder component

The sigmoid function is expressed with the code below:

```
def sigmoid(x):
    return 1 / (1 + torch.exp(-1 * x))
```

The sigmoid function with PyTorch

After rigorously training the model with different hyperparameters, specifically:

- $k \in \{10, 50, 100, 200, 500\}$
- $lr \in \{0.001, 0.005, 0.01, 0.025, 0.05, 0.1\}$
- `num_epochs` $\in \{10, 20, 30, 50, 80, 100\}$

we have come up with $k = 10$, $lr = 0.025$ and `num_epochs` $= 45$ as the most optimal hyperparameters, as if `num_epochs` ≥ 50, the validation accuracy will begin to decrease due to overfitting. We noticed that the validation loss shows a very small increasing trend after about 30 epochs, which is likely due to slight overfitting of the data. In order to handle this problem, we applied L2 Regularization, using this code in the `train` method:

```
# L2 regularization
loss = torch.sum((output - target_train) ** 2) + 0.5 * lamb * model.get_weight_norm()
loss.backward()

train_loss += loss.item()
optimizer.step()
```

L2 Regularization in the `train` method

together with a method to get the normalized weights:

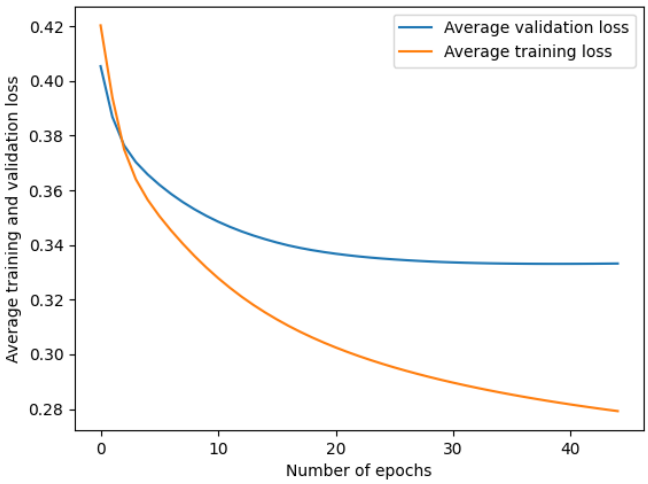
```
def get_weight_norm(self):  1 usage (1 dynamic)
    """ Return ||W^1||^2 + ||W^2||^2.

    :return: float
    """
    g_w_norm = torch.norm(self.g.weight,  p: 2) ** 2
    h_w_norm = torch.norm(self.h.weight,  p: 2) ** 2
    return g_w_norm + h_w_norm
```

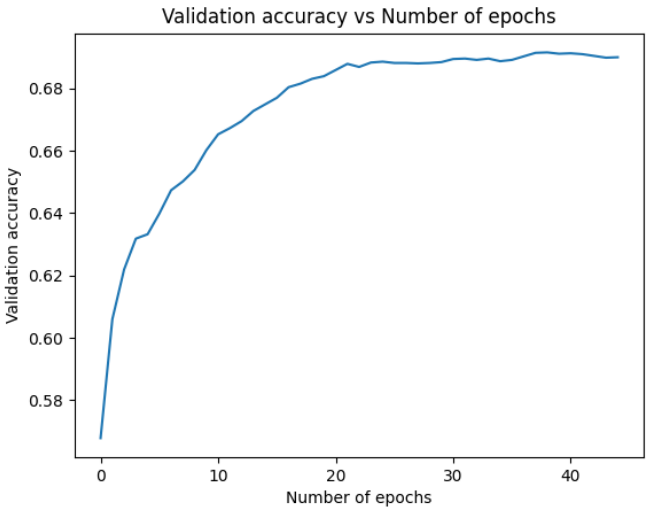
The `get_weight_norm` method

In this case, we had a certain range of regularization penalty (λ) to choose from, specifically between 0.001, 0.01, 0.1 and 1. After testing with all of them,

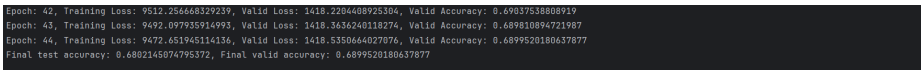
we decided to go for 0.001, since unlike the other values, the problem of overfitting seemed to be resolved since the average validation loss continued to decrease even after 40 epochs. In addition, the final validation accuracy increased to 0.689. As a result, we could conclude that the L^2 regularization improves the model by reducing overfitting. The figures for the training loss, validation loss, validation accuracy with L^2 regularization applied, and the final accuracy values (test and validation) are shown below:



Average training loss and validation loss



Validation accuracy with L^2 regularization applied, with $\lambda = 0.001$



Final accuracy values, with the test accuracy reaching 0.68, and the validation accuracy reaching 0.689

Part B: Algorithm Modification

For this part, we have decided to modify the Item Response Theory (IRT) model. For a complete look at the model, please refer to the code file in the `part_b` folder.

Here are the main takeaways from the enhanced IRT model, compared to the base models:

1. Accuracy Comparison

	Validation Accuracy	Test Accuracy
Baseline Code	0.7058	0.7067
Optimized Code	0.7060	0.7047

Table 1: **Accuracy Comparison Between Baseline and Optimized Code**

Analysis: The optimized version shows a very slight improvement in validation accuracy (from 0.7058 to 0.7060), but the test accuracy slightly drops (from 0.7067 to 0.7047). While this difference is small, it suggests that the optimization techniques did not significantly improve generalization on the test set.

2. Negative Log-Likelihood (NLL) Performance

The optimized code incorporates regularization and mini-batch gradient descent. Regularization helps avoid overfitting, and mini-batch gradient descent can potentially make the optimization process more efficient.

Tracking NLL: The new code tracks negative log-likelihood (NLL) for both the training and validation sets, giving better insight into convergence behavior and overfitting.

3. Regularization Impact

Baseline Code: No regularization. **Optimized Code:** Adds L2 regularization (‘reg_lambda=0.001‘).

Analysis: The regularization is meant to prevent overfitting by penalizing large weights (theta and beta). The results show no drastic change in accuracy, but regularization could still help with robustness, especially with larger data or longer training.

4. Batch Processing

Baseline Code: Processes all data at once (full-batch gradient descent).
Optimized Code: Implements mini-batch gradient descent (batch size = 128).
Analysis: Mini-batch gradient descent typically leads to faster convergence and less noisy updates. Although the overall accuracy didn't improve, mini-batch processing could reduce the training time significantly.

5. Early Stopping

Baseline Code: No early stopping.
Optimized Code: Implements early stopping based on validation NLL, with patience = 10.
Analysis: Early stopping can prevent overfitting, especially when training for a large number of iterations. In this case, it could explain why there's no significant overfitting between training and validation.

6. Visualization of Learning Dynamics

The optimized code introduces detailed plots for negative log-likelihood (NLL) of the training and validation sets, as well as the probability of a correct response vs. student ability (theta). These plots provide more clarity on the learning process and how well the model fits the data.

7. General Conclusion

While the validation accuracy saw a marginal improvement, the test accuracy slightly dropped. The new techniques like regularization, mini-batch gradient descent, and early stopping introduced more flexibility and robustness into the model but did not lead to a significant accuracy improvement over the baseline.

The biggest improvements are in terms of training stability, potential training time reduction (via mini-batch), and better insight into the model through visualization.