CSCI 4170 - Project in AI and ML
Homework 1 - Summer 2025

Haruto Suzuki

Due date: 11:59 pm, May. 29th

Task 1 (20 points): Advanced Objective Function and Use Case

1. Derive the objective function for Logistic Regression using Maximum Likelihood Estimation (MLE). Do some research on the MAP technique for Logistic Regression, include your research on how this technique is different from MLE (include citations).

   For Maximum Likelihood Estimation, we will begin with Bernoulli's Distribution function to represent the loss:

$$L(\theta) = \prod_{n=1}^{N} p^{y_n} \cdot (1-p)^{1-y_n}$$

   where $p = \mathbb{P}(y_n = 1 | x_n, \theta) = \sigma(x_n^T \theta) = \dfrac{1}{1 + e^{-x_n^T \theta}}$, y_n is the label and x_n is the feature for the n^{th} dataset. This loss function is for the Binary Classification. We apply the log and obtain,

$$\begin{aligned} LL(\theta) &= \log \sum_{n=1}^{N} p^{y_n} \cdot (1-p)^{1-y_n} \\ &= \sum_{n=1}^{N} \log p^{y_n} + \sum_{n=1}^{N} \log(1-p)^{1-y_n} \\ &= \sum_{n=1}^{N} y_n \cdot \log p + \sum_{n=1}^{N} (1-y_n) \cdot \log(1-p) \\ &= \sum_{n=1}^{N} y_n \cdot \log p + (1-y_n) \cdot \log(1-p) \end{aligned}$$

   Apply a negative to $LL(\theta)$ to get,

$$NLL(\theta) = -\sum_{n=1}^{N} y_n \cdot \log p + (1-y_n) \cdot \log(1-p)$$

Maximum a Posteriori (MAP) technique for Logistic Regression is another way to derive an objective function, AKA the loss or the cost function, for a model to use to learn parameters. It combines the likelihood of the data with a prior distribution over the model parameters.

Assuming a Gaussian prior over the parameter vector, and applying Bayes' Theorem, we obtain the MAP objective:

$$\mathcal{L}_{MAP}(\theta) = -\sum_{n=1}^{N}[y_n \cdot \log p + (1 - y_n) \cdot \log(1 - p)] + \lambda\theta^T\theta$$

where λ is a hyperparameter coefficient. It is also accurate to have $\frac{\lambda}{2}$ or 2λ instead depending on convention. This is similar to L2 Regularization for logistic regression using L2 norm of a vector.

The key difference between Maximum Likelihood Estimation (MLE) and MAP is that MLE maximizes only the likelihood of the observed data, while MAP also incorporates prior beliefs about the parameters. This makes MAP generally less prone to overfitting, especially in cases with limited data or noisy features.

Source:

- Logistic Regression
- Maximum a Posteriori (MAP) Estimation

2. Define a machine learning problem you wish to solve using Logistic Regression. Justify why logistic regression is the best choice and compare it briefly to another linear classification model (cite your work if this other technique was not covered in class).

In this task, we aim to solve a binary classification problem using logistic regression: predicting whether a credit card client will default on their payment in the next month. The dataset, obtained from the UCI Machine Learning Repository, contains information about 30,000 credit card clients in Taiwan, including demographic variables, past payment history, bill amounts, and previous payments.

Logistic regression is a widely-used and interpretable baseline model for binary classification problems. It estimates the probability that a given input sample belongs to the positive class (in this case, defaulting), making it highly suitable for problems involving credit risk. Since the model outputs calibrated probabilities, it is well-aligned with the goal of assessing a client's likelihood of default.

Key reasons for choosing logistic regression:

- Interpretability: The model's coefficients offer clear insights into feature importance and influence on default risk.
- Efficiency: Logistic regression is computationally efficient, making it suitable for large datasets.

- Probabilistic Output: It provides a probability estimate for default, which aligns with real-world decision thresholds.

Comparison to Support Vector Machine (SVM)

An alternative linear classification model is the Support Vector Machine (SVM) with a linear kernel. While both SVM and logistic regression find a linear decision boundary, their objectives differ. SVM maximizes the margin between the two classes, while logistic regression maximizes the likelihood of the observed labels.

- SVM is robust to outliers in the feature space due to the margin-maximizing principle.
- Logistic regression outputs calibrated probabilities, making it preferable for applications like risk scoring where probability thresholds are meaningful.

According to Ng (2001)[1], logistic regression often outperforms SVMs in low-dimensional problems with large amounts of data, as is the case here.

3. Discuss how your dataset corresponds to the variables in your equations, highlighting any assumptions in your derivation from part 1.

I will be using the **Default of Credit Card Clients** dataset from the UCI Machine Learning Repository. This dataset consists of 30,000 instances and 23 features representing clients' demographic data (e.g., age, education), credit history (e.g., past payment status), and financial attributes (e.g., bill and payment amounts). The goal is to predict whether a client will default on their credit card payment the following month.

Each instance is labeled as:

- `1` for clients who defaulted
- `0` for clients who did not default

Variable Correspondence:

- x_n: Feature vector representing a client's profile (e.g., credit limit, age, repayment history, bill amount).
- y_n: Binary label indicating whether the client defaulted.
- θ: Model parameters learned using gradient descent on the cross-entropy loss.
- $\hat{y}_n = \sigma(x_n^\top \theta + b)$: Predicted probability using the sigmoid function.

Assumptions in Derivation:

- **Independence:** Each client's record is assumed to be independently and identically distributed (i.i.d.).

[1]Ng, A. Y. (2001). On the Discriminative vs. Generative classifiers: A comparison of logistic regression and naive Bayes. *NIPS*

- **Linearity:** The log-odds of default are assumed to be a linear function of the input features.
- **Feature Scaling:** Features are normalized to prevent numerical instability and to ensure fair learning across attributes.
- **Multicollinearity Handling:** Variance Inflation Factor (VIF) is used to detect and mitigate multicollinearity among input features.

Task 2 (20 points): Dataset and Advanced EDA

1. Select a publicly available dataset (excluding commonly used datasets such as Titanic, Housing Prices or Iris). Provide a link to your dataset. Ensure the dataset has at least 10 features to allow for more complex analysis.

   I chose "Default of Credit Card Clients" dataset hosted on UCI Machine Learning Repository.

   Link click here!

2. Perform Exploratory Data Analysis (EDA), addressing potential multicollinearity among features. Use Variance Inflation Factor (VIF) to identify highly correlated variables and demonstrate steps to handle them.

Feature	**VIF**
BILL_AMT2	38.215
BILL_AMT5	35.986
BILL_AMT3	31.783
BILL_AMT4	29.548
BILL_AMT6	21.427
BILL_AMT1	20.844
AGE	11.051
SEX	9.104
EDUCATION	6.731
MARRIAGE	6.286
PAY_5	4.986
PAY_4	4.440
LIMIT_BAL	4.038
PAY_3	3.729
PAY_6	3.464
PAY_2	3.215
PAY_AMT2	2.385
PAY_0	1.997
PAY_AMT3	1.912
PAY_AMT1	1.909
PAY_AMT5	1.854
PAY_AMT4	1.805
default payment next month	1.451
PAY_AMT6	1.271

Table 1: Variance Inflation Factor (VIF) scores for all features

Based on some searching, a Variance Inflation Factor (VIF) value above 5 is generally considered high multicollinear. However, since I don't want to remove majority of the features, I have chosen a threshold of below 7. Features with VIF values exceeding this threshold were removed.

3. Visualize the dataset's feature relationships, ensuring inclusion of at least two advanced visualization techniques (e.g., pair plots with KDE, heatmaps with clustering).

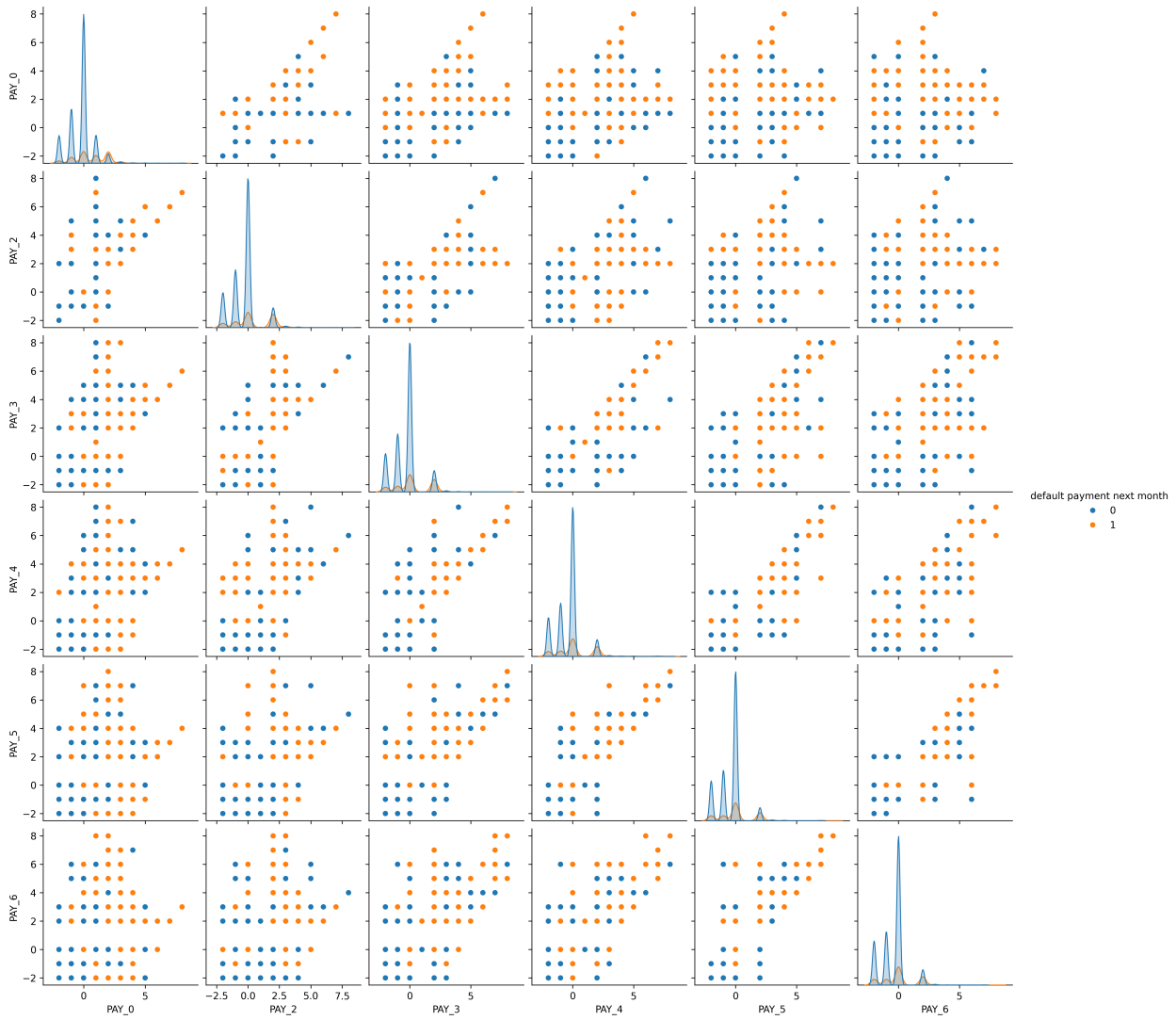

Figure 1: Pair plot of PAY_0 to PAY_6 variables showing repayment patterns

All the data visualized in Figure 1 are integers representing certain states. In the set of PAY_n features, the value represents how late they paid the due amount. There are some correlations we can observe in a few of the plots. For example, PAY_1 and PAY_2 seems to have a linear correlation. Same with PAY_2 and PAY_3, PAY_3 and PAY_4, etc. This is likely because people who didn't pay the this month won't likely pay the next month and this pattern continues. This shows that some of these features are correlated and that it could be removed when training a model. I don't quite see a clear distinction in the distribution of target label that can help with the classification.

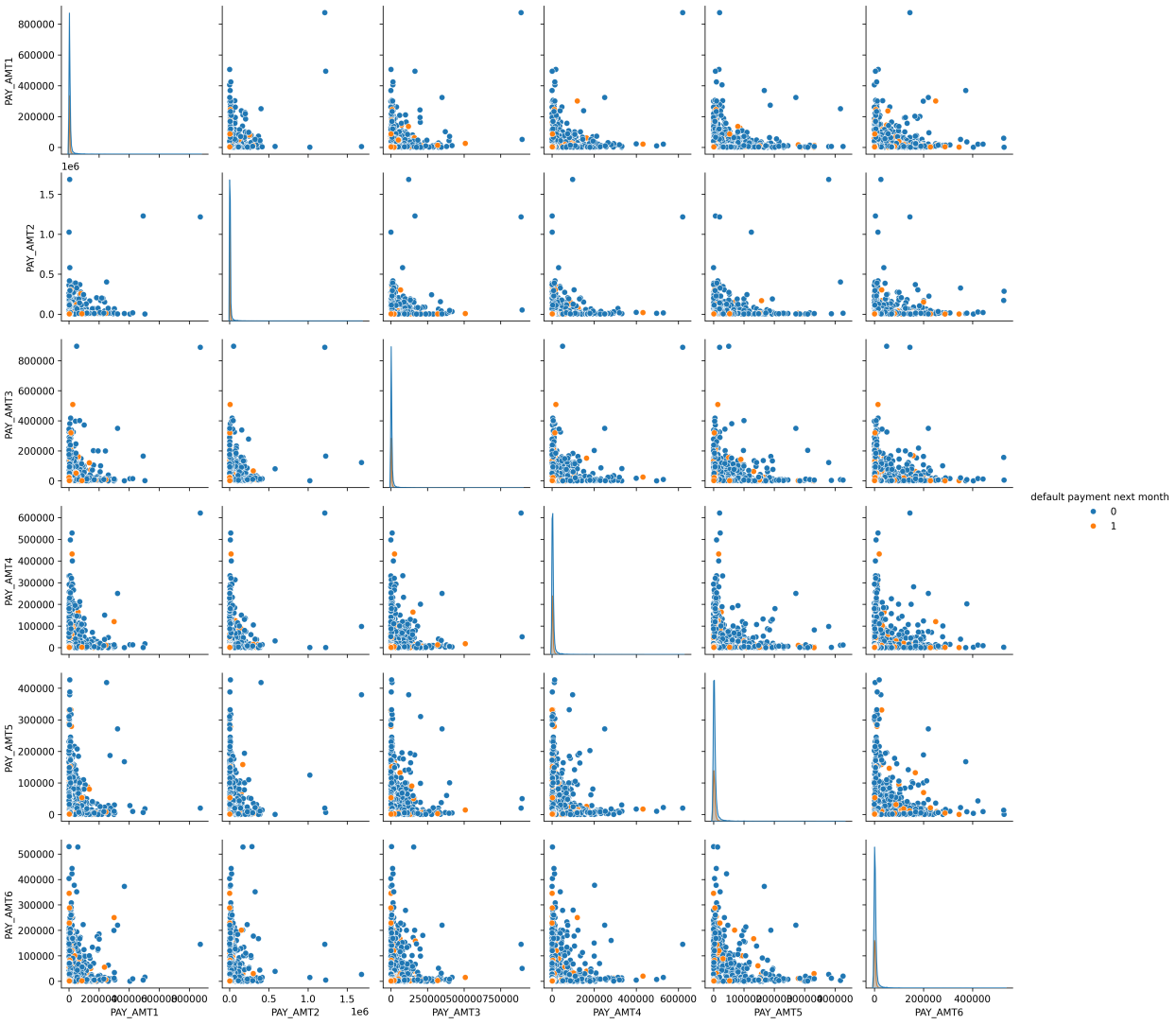

Figure 2: Pair plot of PAY_AMT1 to PAY_AMT6 variables showing payment trends

The data visualized in Figure 2 are numerical values representing the previous payment in Taiwan Dollars. Most of the values are clustered towards the lower region, so it might be challenging to draw a line to classify. Combination of these variables would definitely help as there are no clear linear relationships. There are some outliers or edge cases observed, so there's a challenge on whether to focus on the clustered region or pay attention to the whole space. However, the outlier data appears to be all labeled as 0, so it might be simple for the model to realize that.

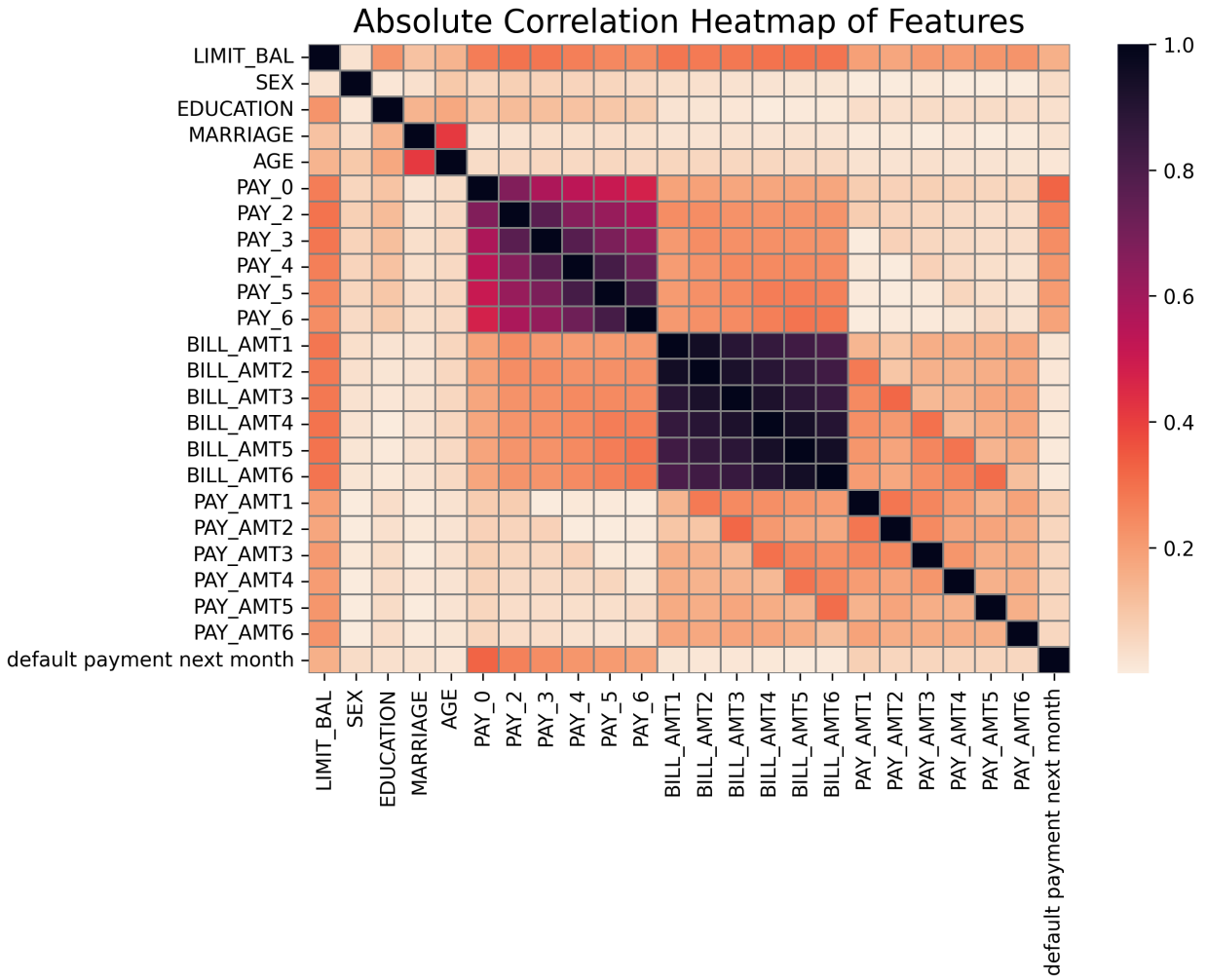

Figure 3: Heatmap of feature correlations showing multicollinearity clusters

Figure 3 is the easier to interpret compared to a pair plot. It is clear that all the BILL_AMT are highly correlated to each other. There are some correlation among the set of PAY_N. On top of that, BILL_AMT are slightly correlated to PAY_N probably because they pay the bill amount listed. Focusing on the target value and its correlation with the features, it seems like PAY_N helps understand the label we want.

Task 3 (20 points): Logistic Regression Implementation

1. Implement Logistic Regression from scratch, including the vectorized implementation of cost function and gradient descent.

   I will show some derivations for the gradient here.

   Linear Combination:

$$\text{Linear Output} = X^\top W + b$$

   Prediction (Sigmoid Output):

$$\hat{y} = \sigma(X^\top W + b)$$

   Loss Function (Binary Cross-Entropy):

$$\begin{aligned}\mathcal{L} &= -\left[y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right] \\ &= -\left[y \log\left(\sigma(X^\top W + b)\right) + (1-y)\log\left(1-\sigma(X^\top W + b)\right)\right]\end{aligned}$$

   Gradient with Respect to the Weights:

$$\frac{\partial \mathcal{L}}{\partial W} = -\left[\frac{y \cdot \sigma'(X^\top W + b)}{\sigma(X^\top W + b)} - \frac{(1-y)\cdot \sigma'(X^\top W + b)}{1-\sigma(X^\top W + b)}\right]$$

   Since:

$$\frac{\partial \sigma(X^\top W + b)}{\partial W} = \sigma(X^\top W + b)(1-\sigma(X^\top W + b)) \cdot X$$

   We get:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W} &= -\left[\frac{y \cdot \sigma(X^\top W + b)(1-\sigma(X^\top W + b)) \cdot X}{\sigma(X^\top W + b)} - \frac{(1-y)\cdot \sigma(X^\top W + b)(1-\sigma(X^\top W + b)) \cdot X}{1-\sigma(X^\top W + b)}\right] \\ &= -\left[y \cdot (1-\sigma(X^\top W + b)) \cdot X - (1-y)\cdot \sigma(X^\top W + b)) \cdot X\right] \\ &= -\left[y \cdot (1-\sigma(X^\top W + b)) - (1-y)\cdot \sigma(X^\top W + b))\right] \cdot X \\ &= -\left[y - y \cdot \sigma(X^\top W + b) - \sigma(X^\top W + b) + y \cdot \sigma(X^\top W + b)\right] \cdot X \\ &= -\left[y - \sigma(X^\top W + b)\right] \cdot X\end{aligned}$$

$$\frac{\partial \mathcal{L}}{\partial W} = (\sigma(X^\top W + b) - y) \cdot X$$

   Gradient with Respect to the Bias:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b} &= -\left[\frac{y \cdot \sigma(X^\top W + b)(1-\sigma(X^\top W + b))}{\sigma(X^\top W + b)} - \frac{(1-y)\cdot \sigma(X^\top W + b)(1-\sigma(X^\top W + b))}{1-\sigma(X^\top W + b)}\right] \\ &= -\left[y - \sigma(X^\top W + b)\right]\end{aligned}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \sigma(X^\top W + b) - y$$

2. Implement and compare the three gradient descent variants (e.g., batch gradient descent, stochastic gradient descent, and mini-batch gradient descent). Explain their convergence properties with respect to your cost function.

   The cost values for each method are shown at selected iterations. The table highlights the progression of the cost function across training epochs for all three methods. The hyperparameters are fixed for all three iterations:

   - Learning rate: 0.1
   - Epoch: 500
   - Threshold: 0.5

Epoch	**Batch GD**	**Stochastic GD**	**Mini-Batch GD**
0	1.8938	0.6121	0.4904
25	1.0792	0.6056	0.4788
50	0.7996	0.6114	0.4145
75	0.6824	0.6017	0.4350
100	0.6194	0.6030	0.3315
125	0.5784	0.6102	0.4523
150	0.5496	0.5995	0.4310
175	0.5289	0.6042	0.4878
200	0.5143	0.6127	0.4205
225	0.5044	0.6044	0.5048
250	0.4975	0.6009	0.3405
275	0.4926	0.6044	0.2792
300	0.4889	0.5996	0.4815
325	0.4860	0.6003	0.3387
350	0.4838	0.6070	0.5750
375	0.4819	0.6062	0.5417
400	0.4803	0.6083	0.3720
425	0.4790	0.6103	0.5374
450	0.4778	0.6042	0.3652
475	0.4767	0.6050	0.6075
499	0.4758	0.6069	0.4652

Table 2: Cost function values over training epochs for each gradient descent method

Method	**Test Accuracy**	**Training Time (s)**
Batch Gradient Descent (BGD)	0.8053	**1.29**
Stochastic Gradient Descent (SGD)	0.8033	287.93
Mini-Batch Gradient Descent (MBGD)	**0.8115**	12.04

Table 3: Final test accuracy and training time for each method

Batch Gradient Descent (BGD): The cost decreases smoothly and consistently, showing a stable convergence trajectory. This is characteristic of BGD, where the gradient is computed over the entire dataset. By the final epoch, the cost settles at 0.4758 with a reliable test accuracy of 0.8053. There are no oscillations or regressions in cost, demonstrating stability.

Stochastic Gradient Descent (SGD): The cost values exhibit mild fluctuations throughout training, showing no clear downward trend. The cost oscillates around 0.60 and does not consistently decrease, indicating slow and noisy convergence. Unlike Batch Gradient Descent or Mini-Batch Gradient Descent, SGD updates the model using single samples, which introduces variance in the learning trajectory. Despite this, the test accuracy (0.8033) remains competitive, suggesting that while the cost function appears less stable, SGD is still capable of finding a reasonably good solution.

Mini-Batch Gradient Descent (MBGD): MBGD exhibits small fluctuations due to batch-level variance but maintains more stable behavior than SGD. The cost values occasionally rise (e.g., from 0.3315 at iteration 100 to 0.4523 at 125), indicating some instability, but overall MBGD provides the most balanced trade-off. It achieves the lowest final cost (0.4652) and the highest test accuracy (0.8115), making it the best-performing method in practice.

Training Time and Performance Trade-off: While all three methods achieve comparable test accuracy, their training times differ significantly. Batch Gradient Descent completes training the fastest (1.29s), but its accuracy (0.8053) is slightly lower than Mini-Batch Gradient Descent, which achieves the highest accuracy (0.8115) in a reasonable time (12.04s). On the other hand, Stochastic Gradient Descen, despite being competitive in accuracy (0.8033), is computationally expensive, taking 287.93 seconds due to its sample-wise updates. Considering both accuracy and training efficiency, Mini-Batch Gradient Descent provides the best trade-off between convergence quality and computational cost.

Task 4 (40 points): Optimization Techniques and Advanced Comparison

1. Implement or use packages to incorporate any three optimization algorithms (e.g., Momentum, RMSProp, Adam). Compare their performance with the vanilla stochastic gradient descent implementation from Task 3.

   I implemented Momentum, RMSProp, and Adam using some online resources. I will compare the performance of each optimization algorithm with Batch Gradient Descent since Stochastic Gradient Descent takes too long to run in general. The hyperparameters are the same as before:

   - Learning rate: 0.1
   - Epoch: 500
   - Threshold: 0.5

Epoch	**BGD (Vanilla)**	**BGD (Momentum)**	**BGD (RMSProp)**	**BGD (Adam)**
0	1.8938	1.8938	1.8938	1.8938
25	1.0792	1.2273	0.4862	0.5068
50	0.7996	0.8435	0.4843	0.4766
75	0.6824	0.7013	0.4757	0.4676
100	0.6194	0.6327	0.4812	0.4663
125	0.5784	0.5879	0.4796	0.4657
150	0.5496	0.5561	0.4715	0.4654
175	0.5289	0.5333	0.4828	0.4653
200	0.5143	0.5171	0.4819	0.4652
225	0.5044	0.5060	0.4752	0.4651
250	0.4975	0.4986	0.4808	0.4651
275	0.4926	0.4934	0.4792	0.4651
300	0.4889	0.4895	0.4712	0.4651
325	0.4860	0.4866	0.4815	0.4651
350	0.4838	0.4843	0.4844	0.4651
375	0.4819	0.4823	0.4752	0.4651
400	0.4803	0.4807	0.4788	0.4651
425	0.4790	0.4793	0.4837	0.4651
450	0.4778	0.4781	0.4730	0.4651
475	0.4767	0.4771	0.4788	0.4651
499	0.4758	0.4762	0.4909	0.4651

Table 4: Comparison of Cost Function over Epochs for Batch Gradient Descent (BGD) Variants

Method	Test Accuracy	Training Time (s)
Batch Gradient Descent (BGD) Vanilla	0.8053	1.68
Batch Gradient Descent (BGD) with Momentum	0.8060	3.33
Batch Gradient Descent (BGD) with RMSProp	0.7872	1.44
Batch Gradient Descent (BGD) with Adam	0.8098	1.43

Table 5: Comparison of Test Accuracy and Training Time for BGD Variants

Based on the selected cost output, BGD with Adam does out perform the other in terms of the cost in almost all printed epochs. We can observe that it reaches a low loss at a high speed compared to the others, indicating the effectiveness of the optimization algorithm. In the end, we see that BGD with Adam achieves the lowest cost. The running time for all but with Momentum are about the same. With the given speed and the accuracy, we can be confident that BGD is efficient with high return model.

Resource:

- GeeksForGeeks: What is Adam Optimizer?
- GeeksForGeeks: Momentum-based Gradient Optimizer - ML
- YouTube Video: Who's Adam and What's He Optimizing?

**If you want to see the output of SGD, I left my previous version below, but it doesn't fully answer the question. SGD takes a long time to run, so I believe it is not relevant to run this experiment multiple times

INCORRECT: So I thought I only had to implement one optimizer. The following is my first version for question 1.

Epoch	SGD without Adam	SGD with Adam
0	0.6121	0.8088
25	0.6056	0.8119
50	0.6114	0.8147
75	0.6017	0.8080
100	0.6030	0.8150
125	0.6102	0.7984
150	0.5995	0.8100
175	0.6042	0.8159
200	0.6127	0.8125
225	0.6044	0.8137
250	0.6009	0.8107
275	0.6044	0.8108
300	0.5996	0.8058
325	0.6003	0.8030
350	0.6070	0.8092
375	0.6062	0.7993
400	0.6083	0.8256
425	0.6103	0.8137
450	0.6042	0.8016
475	0.6050	0.8009
499	0.6069	0.8076

Table 6: Comparison of Cost Function over Epochs for SGD without Adam vs. SGD with Adam

Method	Test Accuracy	Training Time (s)
Stochastic Gradient Descent (SGD) w/o Adam	0.8033	287.93
Stochastic Gradient Descent (SGD) with Adam	0.7875	646.95

Table 7: Comparison of test accuracy and training time for SGD with and without Adam

Based on the selected cost output, we see the cost for SGD with Adam is higher at all times. The final test accuracy is lower with Adam and it takes around twice as much to finish 500 epoch. As for this case, applying Adam is not beneficial in every aspect.

2. Define and use multiple evaluation metrics (e.g., precision, recall, F1 score) to analyze and interpret results for each algorithm.

   To evaluate the performance of each optimization algorithm, I used three key classification metrics: precision, recall, and F1 score. These provide a more nuanced assessment than accuracy, particularly for imbalanced datasets like credit default prediction. Precision measures the ability to avoid false positives, recall assesses the detection of actual positives, and F1 score balances the two. The hyperparameters used are detailed in Task 4, Question 1.

Algorithm	**Precision**	**Recall**	**F1 Score**
Batch Gradient Descent (Vanilla)	0.6600	0.2277	0.3386
Batch Gradient Descent (with Momentum)	0.6637	0.2300	0.3416
Batch Gradient Descent (with RMSProp)	0.6385	0.0632	0.1150
Batch Gradient Descent (with Adam)	0.6920	0.2361	**0.3521**

Table 8: Comparison of Precision, Recall, and F1 Score for Batch Gradient Descent Variants

   Among the Batch Gradient Descent variants, the Adam optimizer achieved the highest F1 score (0.3521), offering the best trade-off between precision (0.6920) and recall (0.2361). While the vanilla BGD and BGD with Momentum had slightly lower F1 scores (0.3386 and 0.3416 respectively), their performance was quite close, indicating consistent improvement from adding momentum. BGD with RMSProp, despite a reasonable precision (0.6385), had extremely low recall (0.0632), resulting in a significantly weaker F1 score (0.1150). It is interesting as from Table 5, we see that there aren't much difference between the accuracies, but in terms of how accurate the model classifies each label separately, there's a significant difference. These findings suggest that Adam provides the most balanced and reliable optimization for batch training in this context, especially when F1 score is the priority.

   **If you would like to see me analyzing three gradient descent algorithm with Adam optimizer, I left it below as well.

   INCORRECT: So I thought I only had to implement one optimizer and compare to the three gradient descent algorithm. The following is my first version for question 1.

   Among the three algorithms tested with the Adam optimizer, Stochastic Gradient Descent (SGD) achieved the highest F1 score (0.4558), reflecting a better balance between precision (0.5184) and recall (0.4067). Batch Gradient Descent had the highest precision (0.6920) but low recall (0.2361), resulting in a lower F1 score (0.3521). Mini-Batch Gradient Descent showed similarly high precision (0.6698) but had the weakest recall (0.1097), leading to the lowest F1 score (0.1885). These results suggest that while batch and mini-batch methods were more conservative, stochastic descent was more effective at identifying defaulters—making it preferable when optimizing for F1.

Algorithm	Precision	Recall	F1 Score
Batch Gradient Descent (with Adam)	0.6920	0.2361	0.3521
Stochastic Gradient Descent (with Adam)	0.5184	0.4067	**0.4558**
Mini-Batch Gradient Descent (with Adam)	0.6698	0.1097	0.1885

Table 9: Comparison of precision, recall, and F1 score for each optimization algorithm using the Adam optimizer.

One possible reason for SGD's superior F1 score is that it updates model parameters after each individual example, allowing it to adapt quickly to rare patterns in the data. This is advantageous for imbalanced datasets, where minority class instances (e.g., defaulters) are underrepresented. In contrast, batch and mini-batch methods average gradients over larger subsets, which can dilute the impact of less frequent labels and reduce recall.

3. Perform a hyperparameter tuning process (manual or automated using grid search/random search) for each optimization algorithm and assess its impact on performance. If you have to do some research for these techniques, please cite your sources.

To evaluate the impact of optimization algorithms on logistic regression performance, I conducted a manual grid search across hyperparameters including learning rate, number of epochs, and use of the Adam optimizer. To streamline the process of trying all combinations, I searched for and used a Python function that automatically generated Cartesian products (i.e. using itertools.product). This allowed me to systematically explore different configurations for each epoch size, learning rate, and optimizer. I chose Batch Gradient Descent for this execution since it takes the least amount of time and gives a good return.

The hyperparameter grid search for batch gradient descent revealed that the RMSProp optimizer with a learning rate of 0.001 and 1000 epochs achieved the highest F1 score of 0.3956, along with the accuracy of 0.7607. It outperformed Adam and Momentum in terms of F1 score, although Adam with a learning rate of 0.01 and 100 epochs followed closely with an F1 score of 0.3874 and significantly lower training time (0.28 seconds). While the Vanilla batch gradient descent model was relatively efficient (1.39 seconds), its F1 score of 0.3637 lagged behind the optimizer-enhanced variants. Overall, the search demonstrates that using an optimizer, especially RMSProp or Adam, can substantially improve classification performance in batch settings, with a trade-off between computational cost and predictive accuracy. It is interesting to see that although there are lots of combinations that accomplished higher accuracy, the one with the somewhat lower accuracy performs the best in terms of how well it classifies each label separately.

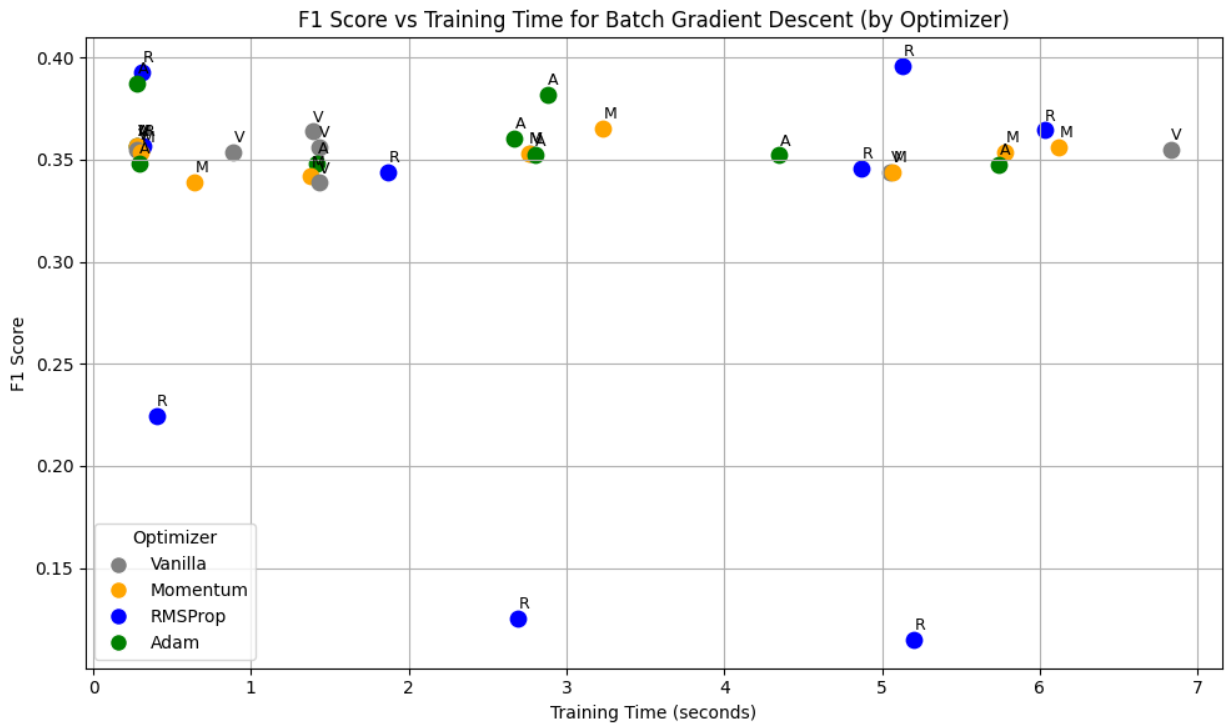

Figure 4: Scatter plot of F1 score versus training time for different optimization algorithm

I plotted F1 Score vs Time in Figure 4 because F1 Score captures the true accuracy in term of True Positive (TP) and True Negative (NP), which is more descriptive of what the model is good at labeling. We can see that all the combinations have finished training in less than 7 seconds which is extremely efficient. In contrast with Stochastic Gradient Descent, it takes a few hours just getting through some configuration. Among the optimizers, RMSProp exhibited the most fluctuation in F1 score across different hyperparameter settings, whereas Vanilla, Momentum, and Adam maintained more consistent performance in the 0.35 F1 range. This suggests that while RMSProp can yield the highest scores under optimal conditions, it may require careful tuning to consistently achieve strong results.

**Grid Search of three gradient descent method and its analysis below.

INCORRECT: I thought I only had to implement one optimizer and do a grid search across the three gradient descent algorithm implemented. The following is my first version for question 1.

The hyperparameter search revealed that mini-batch gradient descent with Adam (learning rate = 0.01, 500 epochs) achieved the highest F1 score of 0.4049, with a training time of just 22.14 seconds. In contrast, stochastic gradient descent without Adam had slightly lower F1 performance (0.3676) but took significantly longer—nearly 300–650 seconds depending on the configuration. Vanilla batch gradient descent, while fast (5.54 seconds), lagged in F1 score (0.3818), suggesting it may not generalize as well

in imbalanced classification tasks. Overall, the grid search process highlighted that enabling Adam generally improves performance, especially when paired with mini-batch descent, but comes at the cost of higher computational time for stochastic variants.

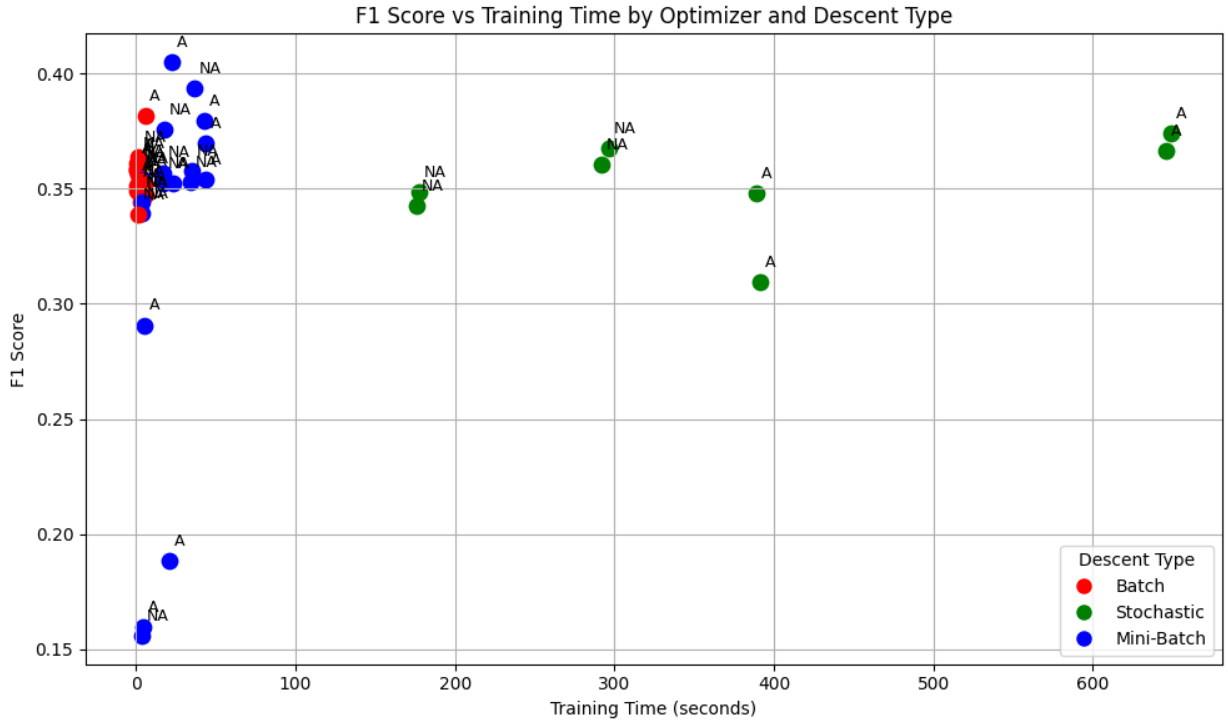

Figure 5: Scatter plot of F1 score versus training time for different gradient descent method with adam optimizer

From this figure, it is clear that Batch GD takes significatly the least amount of time compared to Stochastic and most of Mini-Batch. Batch GD produced a good F1 Score given the time taken. Mini-Batch GD showcased a significant gap between hyperparameter configurations. Overall, Mni-Batch performed the best. Stochastic GD has a steady F1 score between 0.30 and 0.38, but it takes significantly more time.

4. Conclude by discussing the practical trade-offs of the algorithms, including computational complexity, interpretability, and suitability for large-scale datasets.

The batch gradient descent with the optimizers each presents practical trade-offs. Vanilla gradient descent is the most interpretable and computationally lightweight, making it ideal for small-scale problems. Momentum adds minimal complexity while improving convergence speed and stability. RMSProp and Adam, though more computationally intensive and less interpretable due to adaptive learning rates and internal state tracking, generally achieve better performance on complex or noisy data.

However, all batch methods require computing gradients over the entire dataset for each update, which limits their scalability to large datasets. This can lead to more

memory and slower iterations compared to stochastic or mini-batch methods. In practice, Adam is often the best balance of speed and accuracy for medium-sized datasets, while Vanilla and Momentum may be preferred when simplicity, reproducibility, or low-resource environments are prioritized.

**Version 1 summary below

INCORRECT: I thought I only had to implement one optimizer. Thus I did the final analysis based on all three gradient descent method with Adam. The following is my first version for question 1.

When comparing the three gradient descent methods, Batch Gradient Descent, Stochastic Gradient Descent (SGD), and Mini-Batch Gradient Descent, each presents distinct trade-offs in terms of computational complexity, interpretability, and scalability. Batch Gradient Descent is computationally efficient per epoch and easy to interpret, as it computes gradients using the entire dataset, resulting in smooth and stable convergence. However, its memory demands and inability to update parameters until a full pass through the data make it less suitable for large-scale datasets.

Stochastic Gradient Descent, especially with Adam, is highly adaptable and performs well on imbalanced datasets due to its frequent updates, but this comes with high variance in parameter updates and significantly longer training times. Mini-Batch Gradient Descent strikes a balance between stability and efficiency by updating weights based on small subsets of the data. It is generally more scalable than full-batch methods and more stable than SGD, making it a practical choice for large-scale machine learning tasks. Ultimately, the choice of optimizer depends on the problem context—SGD may be preferred for higher sensitivity in rare-event detection, while mini-batch approaches offer better speed-performance trade-offs in production environments.