NYCU Introduction to Machine Learning, Homework 2

[111550196], [狄豪飛 Jorge Tyrakowski]

The screenshot and the figures we provided below are just examples. **The results below are not guaranteed to be correct.** Please ensure your answers are clear and readable, or no points will be given. Please also remember to convert it to a PDF file before submission. You should use English to answer the questions. After reading this paragraph, you can delete this paragraph.

Part. 1, Coding (60%):

(25%) Logistic Regression w/ Gradient Descent Method

1. (5%) Show the hyperparameters (learning rate and iteration, etc) that you used and the weights and intercept of your model.

```
LR = LogisticRegression(
    learning_rate=0.008,  # You can modify the parameters as you want
    num_iterations=1000,  # You can modify the parameters as you want
)
```

```
(/home/dihaofei/conda-envs/ML-NYCU) dihaofei@dihaofei:~/NYCU/Machine Learning/HW02$ python3 main.py
(166,)
2024-10-26 21:00:39.546 | INFO     | __main__:main:294 - LR: Weights: [-0.08115022  0.02979435  0.11088106  0.0945041   0.06750898],
Intercep: -0.32982877884705314
```

2. (5%) Show the AUC of the classification results on the testing set.

```
AUC=0.8773
```

3. (15%) Show the accuracy score of your model on the testing set

```
LR: Accuracy=0.8333
```

(25%) Fisher Linear Discriminant, FLD

4. (5%) Show the mean vectors m_i (i=0, 1) of each class, the within-class scatter matrix *Sw,* and the between-class scatter matrix *Sb* of the training set.

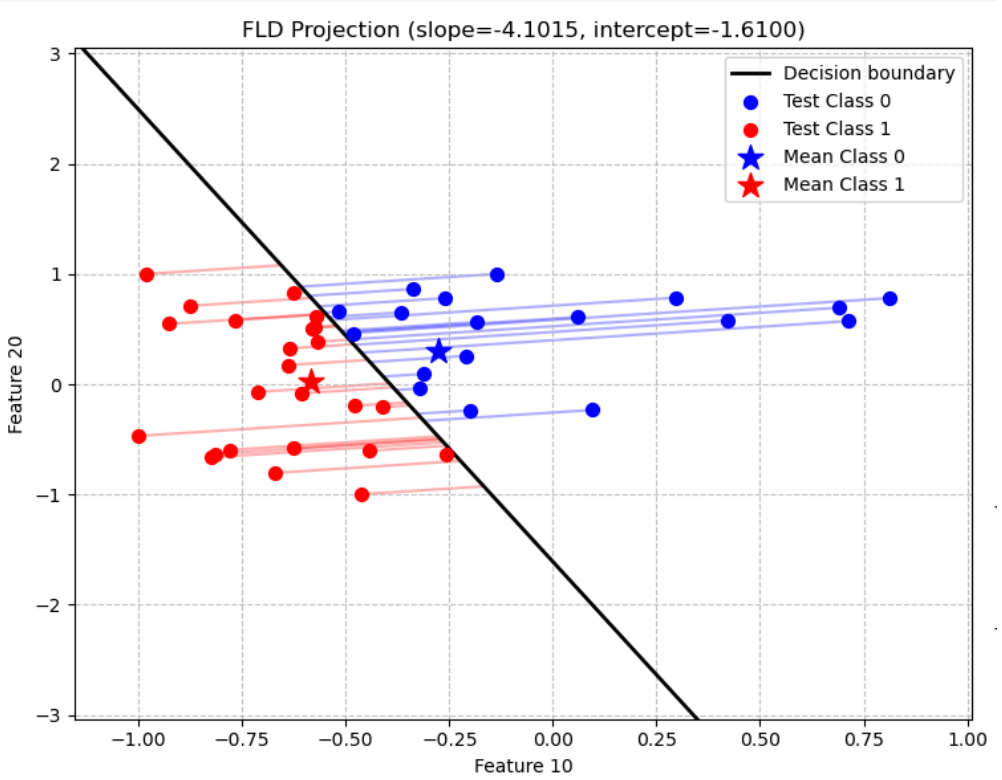
```
2024-10-26 21:00:39.548 | INFO     | __main__:main:320 - FLD: m0=[-0.27747695  0.29565197], m1=[-0.58535466  0.02331584]
 of cols=['10', '20']
2024-10-26 21:00:39.549 | INFO     | __main__:main:321 - FLD:
Sw=
[[17.17974856  5.44299487]
 [ 5.44299487 44.81848741]]
2024-10-26 21:00:39.549 | INFO     | __main__:main:322 - FLD:
Sb=
[[0.09478869 0.08384622]
 [0.08384622 0.07416696]]
```

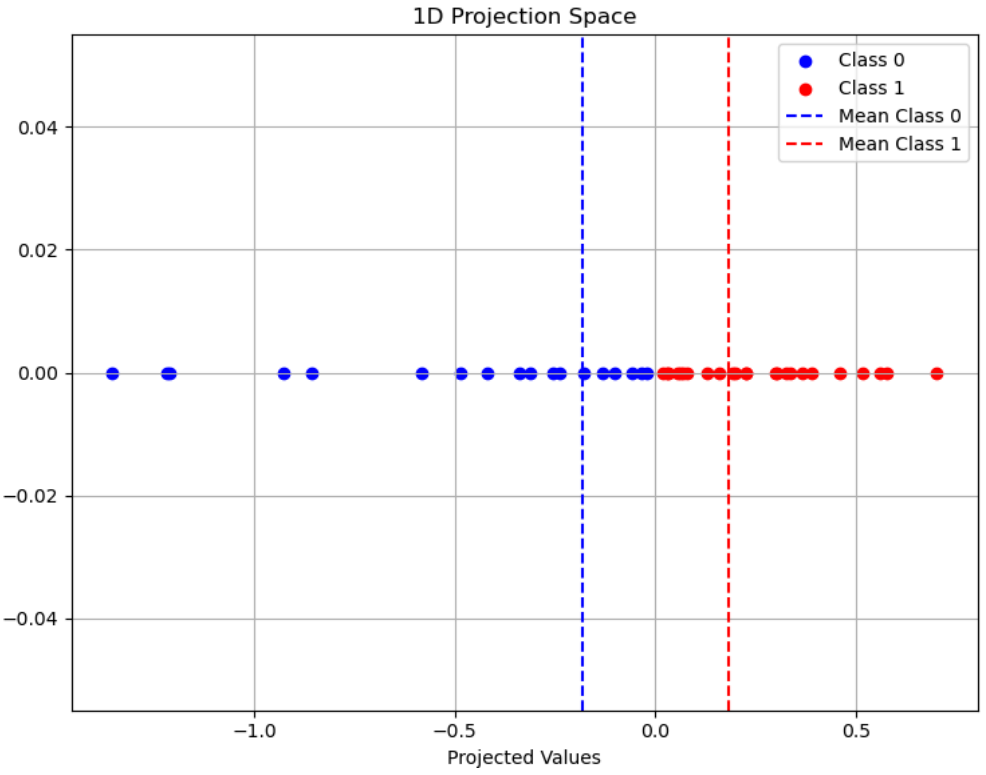
5. (5%) Show the Fisher's linear discriminant *w* of the training set.

```
2024-10-26 21:00:39.549 | INFO     | __main__:main:323 - FLD:
w=
[-0.97154001 -0.23687549]
```

6. (15%) Obtain predictions for the testing set by measuring the distance between the projected value of the testing data and the projected means of the training data for the two classes. (Also, plot for training data). Show the accuracy score on the testing set.

```
FLD: Accuracy=0.7619
```





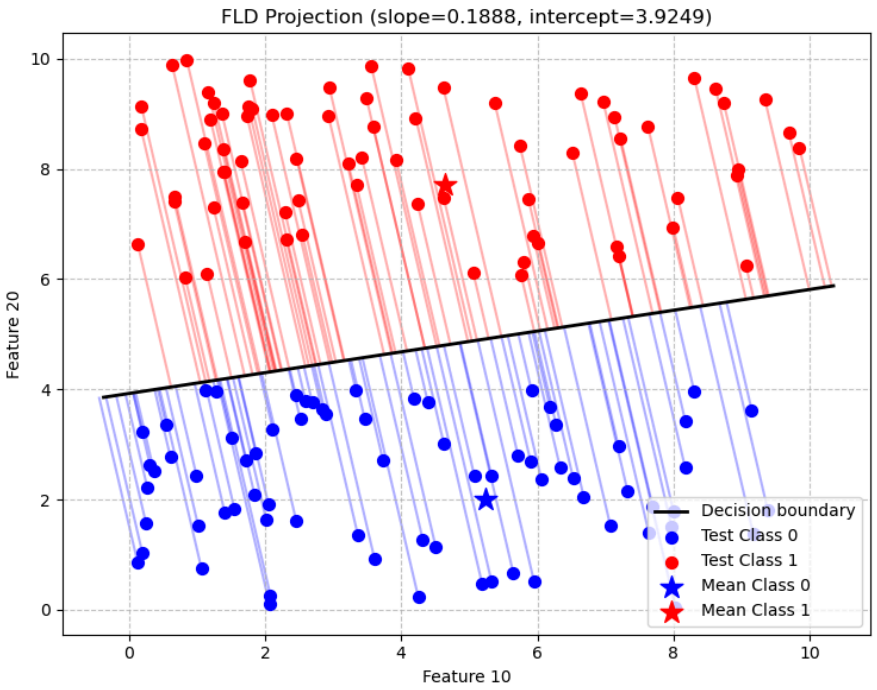
(10%) Code Check and Verification

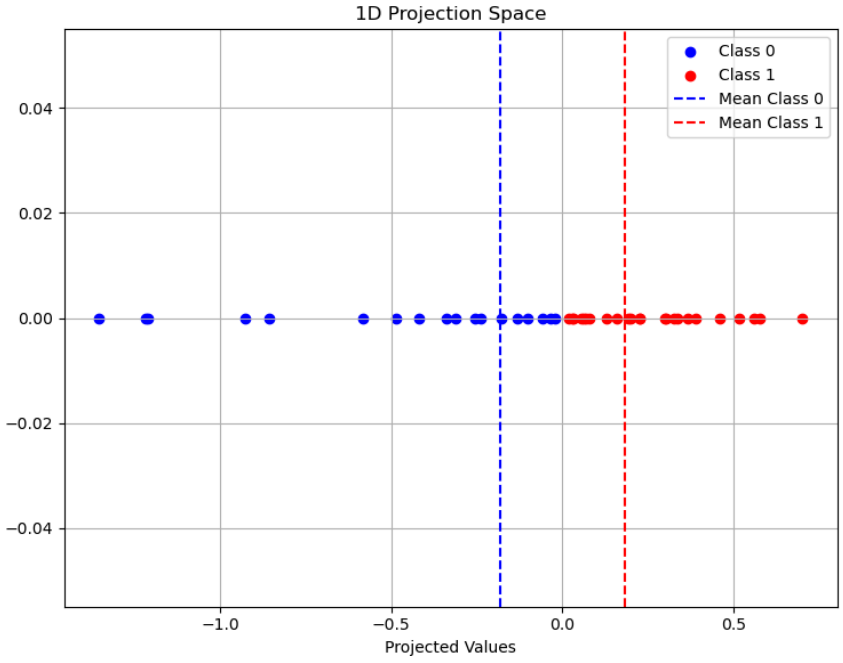
7. (10%) Lint the code and show the PyTest results.

```
dihaofei@dihaofei: ~/NYCU/Machine Learning/HW02
(/home/dihaofei/conda-envs/ML-NYCU) dihaofei@dihaofei:~/NYCU/Machine Learning/HW02$ flake8 main.py
(/home/dihaofei/conda-envs/ML-NYCU) dihaofei@dihaofei:~/NYCU/Machine Learning/HW02$
```

```
(/home/dihaofei/conda-envs/ML-NYCU) dihaofei@dihaofei:~/NYCU/Machine Learning/HW02$ pytest test_main.py -s
============================================= test session starts ==============================================
platform linux -- Python 3.12.5, pytest-7.4.4, pluggy-1.0.0
rootdir: /home/dihaofei/NYCU/Machine Learning/HW02
collected 2 items

test_main.py (395, 2) (395,)
2024-10-26 21:14:43.915 | INFO     | test_main:test_logistic_regression:35 - accuracy=0.9517
.(395, 2) (395,)
2024-10-26 21:14:43.917 | INFO     | test_main:test_fld:45 - accuracy=0.8759
```





Part. 2, Questions (40%):

1. **(10%)**
   - **Is logistic 'regression' used for regression problems?**
   - **If not, what task is it primarily used? (without any additional techniques and modification); If yes, how can it be implemented?**
   - **Why are we using the logistic function in such a task? (list two reasons)**
   - **If there are multi-class, what should we use to substitute it?**

Despite its potentially misleading name, **Logistic "Regression"** is **not** actually used **for regression problems. Instead**, it serves as a powerful method **for classification tasks**. While regression models aim to predict continuous values (such as student performance, house prices, or stock prices), logistic regression **is specifically designed to solve binary classification problems** where we need to categorize inputs into one of two classes. For example, it's commonly used to determine if a tumor is malignant or benign, or if a student will pass or fail a course based on their study hours.

The logistic function, also known as the **sigmoid function**, plays a crucial role in this algorithm for two fundamental reasons. **First**, it elegantly **transforms any input value into a probability between 0 and 1**, which is exactly what we need for **binary classification.** This means that no matter how extreme our input values might be, the function will always give us a valid probability that we can use to make our classification decision. For instance, if our model outputs 0.9, we can confidently say there's a 90% chance that the input belongs to the positive class. **Second,** the logistic function has special mathematical properties that make it particularly useful for machine learning. **It's smooth and differentiable**, which means **we can easily calculate gradients for optimization, and its S-shaped curve naturally mirrors the way we make binary decisions** - when evidence strongly points in one direction, we're very confident in our classification, but in borderline cases, small changes in input can significantly affect our decision.

When we need to handle problems with more than two classes (like classifying handwritten digits from 0 to 9, or categorizing images of different animals), we need to modify our approach. In these cases, **we replace the logistic function with what's called the Softmax function**. The **Softmax function** is essentially a generalization of the logistic function that **can handle multiple classes**. It takes a vector of numbers and transforms them into a probability distribution over all possible classes. This means it maintains the key benefits of the logistic function (like outputting values between 0 and 1) while allowing us to work with multiple categories. For example, when classifying an image of an animal, the Softmax function might tell us there's a 70% chance it's a dog, a 20% chance it's a wolf, and a 10% chance it's distributed among other animal categories.

2. **(15%) When a trained classification model shows exceptionally high precision but unusually low recall and F1-score, what potential issues might arise? How can these issues be resolved? List at least three solutions.**

When a trained classification model shows **exceptionally high precision** but **unusually low recall and F1-score**, we're facing an important problem that needs careful attention. Imagine a bridge monitoring system using sensors to detect potential structural problems. High precision means that when our system warns about a problem, it's usually correct - maintenance teams aren't wasting time checking false alarms. However, low recall means it's missing many actual problems, perhaps only

detecting obvious major cracks while missing early warning signs like small stress patterns that could prevent future collapses. Similarly, think of a cautious doctor who only diagnoses patients as sick when absolutely certain. While they rarely misdiagnose healthy people as sick (high precision), they might miss many patients who actually need treatment (low recall). The low F1-score in these situations is particularly telling because it shows us that our model isn't well-balanced. Even though precision is high, the F1-score combines precision and recall into a single metric that helps us see the overall performance. When there's a big gap between precision and recall, the F1-score suffers, warning us that our model isn't performing as well as we might think if we only looked at precision.

This situation creates three main problems. **First**, we're missing many important cases that need attention - in our bridge example, this means overlooking potential structural failures that could become dangerous. **Second**, our model has become too cautious, like that overly careful doctor who only diagnoses the most obvious cases. **Third**, this can give us a false sense of security: since the model is rarely wrong when it raises an alarm, we might think it's working perfectly, not realizing it's missing many critical cases. Here are four detailed solutions to fix these issues:

1. **Adjust the Decision Threshold:** Imagine our model gives each case a score from 0 to 1, where 1 means "definitely positive." By default, it only says "yes" if the score is above 0.5. We can make our model less cautious by lowering this threshold - maybe to 0.3. This means it will say "yes" even when it's only 30% sure, helping catch more potential problems. In our bridge example, this would mean flagging more potential issues for inspection, even if we're not completely certain they're problems.
2. **Balance the Training Data:** Often, this problem happens because we trained our model with unbalanced data - maybe we had many examples of normal bridge conditions but few examples of problems. We can fix this using special techniques like SMOTE, which creates new, realistic examples of the rare cases. It's like giving our model more practice with the important but rare situations it needs to recognize.
3. **Create a Two-Stage System:** Instead of trying to make a perfect single decision, we can use two steps. The first step is more sensitive, catching anything that might be a problem. The second step looks more carefully at these flagged cases. It's like having a junior doctor do initial screenings (being more willing to flag potential issues) before sending suspicious cases to specialists. This helps us catch more problems while still maintaining good precision.
4. **Make Errors Cost Different Amounts:** We can teach our model that missing a real problem (false negative) is worse than raising a false alarm (false positive). In bridge monitoring, failing to detect a structural weakness is much more serious than sending a maintenance team to check something that turns out to be fine. We build this understanding directly into how the model learns, making it naturally more careful about missing potential problems.

For example, in bridge monitoring with these solutions implemented:

- The lower threshold means we inspect more potential issues
- Balanced training data helps recognize various types of structural problems
- The two-stage system might use simple sensors first, then detailed analysis for suspicious cases
- Different error costs ensure the model prioritizes safety over maintenance efficiency

3. **(15%) In this homework, we use Cross-Entropy as the loss function for Logistic Regression. Can we use Mean Square Error (MSE) instead? Why or why not? Please explain in detail.**

When we're doing **Logistic Regression**, choosing the right **loss function** is crucial. We could use **Mean Square Error (MSE)**, as it's a common and intuitive way to measure errors, but there's a good reason why we prefer **Cross-Entropy** instead. Let me explain why through both the math and some practical examples.

Think about what our logistic regression model is doing: it uses a **sigmoid function** to give us **probabilities** between 0 and 1. When we combine this sigmoid function with MSE, something interesting but problematic happens. The main issue isn't that MSE makes our problem non-convex (it actually stays convex in many cases), but rather that it makes learning really **inefficient**, especially when our model is making big mistakes.

Here's what's happening under the hood: when we use MSE, the **gradient** becomes tiny exactly when we need it most. Mathematically, this happens because the **gradient of MSE** with the sigmoid function is proportional to $\hat{y}(1-\hat{y})(\hat{y}-y)$. Let's break this down: as our prediction $\hat{y}$ gets close to either 0 or 1 (which happens at the flat parts of the sigmoid curve), the term $\hat{y}(1-\hat{y})$ becomes very small. For example, if $\hat{y} = 0.01$, then $\hat{y}(1-\hat{y}) = 0.01 * 0.99 = 0.0099$, giving us a **tiny gradient** even if our prediction is completely wrong. This is like trying to correct someone who's confidently wrong by whispering - not very effective.

Cross-Entropy, on the other hand, behaves much more sensibly. When our model is way off, Cross-Entropy provides a **strong learning signal**. Its gradient is simply proportional to $(\hat{y}-y)$, which means the correction is directly related to how wrong we are. If we predict $\hat{y} = 0.1$ when the true value is $y = 1$, we get a **large gradient** of 0.9, pushing our model to correct this error quickly.

But there's an even deeper reason why Cross-Entropy makes more sense. Remember, in Logistic Regression, we're trying to predict **probabilities**. Cross-Entropy is specifically designed to measure differences between **probability distributions**, which is exactly what we're working with. MSE, in contrast, is better suited for predicting **continuous values** like temperatures or prices. Using MSE for probabilities is a bit like using a ruler to measure weight - it could work, but it's not the right tool for the job.

So while we could use MSE, Cross-Entropy is simply a better choice for Logistic Regression. It helps our model **learn faster** when it's wrong by providing **stronger gradients** exactly when we need them, aligns better with what we're trying to predict (probabilities), and ultimately leads to more **efficient** and **effective training**.