CS5011-A4
REPORT
220026989

1. Introduction

This document provides an overview of Practical 4 in CS5011, which focuses on addressing the Automated Algorithm Selection (AS) problem using artificial neural networks (ANNs). The AS problem entails selecting the most suitable algorithm from a collection of options to solve a given problem instance, and ANNs provide an effective method for automating this selection process. The report will provide a concise discussion of the design evaluations conducted during the practical.

1.1Checklist

- ✓ Part 1
- ✓ Part 2 (Basic)
- ✓ Part 3 (Advanced)
- ✓ Part 3 (Extension 2) - Training done
- ✓ Hyperparameter tuning

1.2 Executing

- ./install.sh
- ./run.sh

2.Design and Implementation

1. Project Structure

- **Main folder : script**
  - **P1 (Folder)**
    - evaluate.py
    - regression.py
  - **P2_1(Folder)**
    - evaluate.py
    - P2_1_classification.py
  - **P2_2(Folder)**
    - evaluate.py
    - P2_2_cost.py

        - **Hyperparameter_tuning**
            - Hyperparameters_regression.txt (best hyperparameters after tuning)
            - Classification_2_tuning.py (tuning hyperparameters for classification_advanced).
            - Classification_tuning.py (tuning hyperparameters for classification_basic).
            - multi_target_regression.py (tuning hyperparameters for regression)
            - random_forest.txt (best hyperparameters for rf)
            - rf.py
        - **models**
            - **Contains saved models and results**
        - **random_forest.py**
        - **train.py**
    - **plots**
    - **results**

2. Codes explored

a. train.py

This code is a Python script that takes command-line arguments using argparse and runs different functions depending on the specified arguments. The functions are imported from different Python files, specifically, regression.py, P2_1_classification.py, P2_2_cost.py, and random_forest.py. The code first defines the required arguments for the script, which includes the type of model to be trained, the dataset to be used, and the path where the trained model should be saved.

After parsing the arguments, the code runs a conditional block that determines which function to call based on the value of the model_type argument. If the model type is regression_nn, it calls the regression_model function from regression.py, if the model type is classification_nn, it calls the classification_model function from P2_1_classification.py, if the model type is classification_nn_cost, it calls the classification_cost_model function from P2_2_cost.py, and if the model type is random_forest, it calls the random_forest function from random_forest.py.

Finally, the code prints a message indicating that training is finished. This script can be used to train different types of machine learning models and save them to a specified file.

b. Hyperparameter tuning – multi_target_regression.py.

This code performs hyperparameter tuning for a regression model using the Optuna library. Here's a breakdown of what the code does and why it's required:

1. Load and preprocess the data:
    - The code reads input data from an instance file (instance_file) and target algorithm data from a performance file (performance_file).
    - The loaded data is converted into tensors (X_data and y_data).
    - Null values and duplicates are checked in the data.
2. Define the dataset and split into training and validation sets:
    - The data is encapsulated in a custom dataset class (ClassificationDataset) that inherits from PyTorch's Dataset class. This allows convenient handling of the data during training and validation.
    - The dataset is split into training and validation sets using random_split, with a specified ratio (80% training, 20% validation).
3. Standardize the data:
    - The training data is standardized by subtracting the mean and dividing by the standard deviation.
    - The same standardization parameters (mean and standard deviation) are applied to the validation data.
4. Define the regression model:
    - The RegressionModel class is defined as a subclass of nn.Module.
    - It defines the architecture of the model with input, hidden, and output layers. ReLU activation is applied after each hidden layer.
    - The model's forward pass is implemented, passing the input through each layer sequentially.
5. Define the training objective function:
    - The objective function is defined, which takes a set of hyperparameters as input and trains the model using those hyperparameters.
    - The function sets up the data loaders for training and validation using the defined datasets.
    - It creates an instance of the regression model and defines the optimizer.

    - The model is trained for a specified number of epochs, with the training loss printed for each epoch.
    - After training, the model is evaluated using the validation set, and the validation loss is calculated.
6. Perform hyperparameter tuning:
    - An Optuna study object is created, specifying the optimization direction as minimization.
    - The objective function is used as the optimization target.
    - The study.optimize method runs the optimization process, performing a specified number of trials (30 in this case) to find the best set of hyperparameters.
    - The best hyperparameters found during the optimization process are printed.
7. Save the best hyperparameters:
    - The best hyperparameters found during the optimization process are saved to a JSON file (hyperparameters.txt) using the json.dump method.
8. Print completion message:
    - The code prints a completion message indicating that the hyperparameter tuning process has finished.

The purpose of this code is to automate the search for optimal hyperparameters for the regression model. By systematically exploring different combinations of hyperparameters, it helps identify the configuration that yields the best performance on the validation set. This is important for optimizing the model's accuracy and generalization capability.

C. hyperparameter_tuning - rf.py

The provided code implements an optimization process for hyperparameter tuning using Optuna library. The objective is to maximize the accuracy of a RandomForestClassifier model for automated algorithm selection. Here's a basic report on the code:

1. Data Loading: The code loads the training data from two separate files: "instance-features.txt" and "performance-data.txt".

2. Objective Function: The objective function is defined, which takes hyperparameters as input and returns the accuracy of the RandomForestClassifier model.

3. Hyperparameter Tuning: The Optuna library is used to create a study object for optimizing the objective function. The study aims to maximize the accuracy by finding the best combination of hyperparameters. The hyperparameters being tuned are pca_components, depth, and n_estimators.

4. Data Preprocessing: The data is split into training and validation sets using train_test_split. StandardScaler is applied to normalize the features. PCA is performed to reduce the dimensionality of the data.

5. Model Training and Evaluation: The RandomForestClassifier model is trained on the training data transformed by PCA. The class_weight is set to "balanced" to handle imbalanced datasets. The trained model is then used to make predictions on the validation set, and the accuracy is calculated using the accuracy_score function.

6. Optimization Process: The objective function is called multiple times by Optuna's optimization algorithm (500 trials in this case) to find the hyperparameter combination that maximizes the accuracy.

7. Best Hyperparameters: After the optimization process, the best hyperparameters found by Optuna are printed and saved to a text file named "random_forest.txt".

8. Conclusion: The code concludes by printing a message indicating the completion of the hyperparameter tuning process.

   Overall, this code performs automated hyperparameter tuning using Optuna for a RandomForestClassifier model, aiming to maximize accuracy in automated algorithm selection.

Note: The hyperparameters for other files only vary slightly by their range and loss functions. The loss functions with their epochs are plotted and saved in plot file.

d. P1 – regression.py

1. Dataset Class (ClassificationDataset):

   - This class is responsible for creating a custom dataset that will hold the input data and target labels for classification.

   - The __init__ method initializes the dataset by taking input data (input) and target algorithm data (algos) as parameters.

   - The __len__ method returns the length of the dataset, which is the number of samples.
   - The __getitem__ method retrieves an item from the dataset at a given index. It returns a tuple containing the input data and target algorithm corresponding to that index.
2. Regression Model Class (Regression):
   - This class defines the architecture of the regression model.
   - The __init__ method initializes the model by taking the input size, output size, and hidden layer size as parameters.
   - The forward propagation through the model is defined in the forward method. It performs a series of linear transformations and applies ReLU activation functions between the layers.
3. Validation Function (validate):
   - The validate function is used to evaluate the trained model using the validation dataset.
   - It takes the trained model, the validation data loader (val_loader), and the loss criterion (criterion) as inputs.
   - Inside the function, the model is set to evaluation mode (model.eval()), and gradients are disabled using torch.no_grad() to speed up the evaluation.
   - It iterates over the validation data and performs a forward pass through the model (outputs = model(X)).
   - The loss between the predicted outputs and the target labels is calculated using the provided loss criterion (loss = criterion(outputs, y)).
   - The average validation loss is computed by summing up the individual losses and dividing by the number of batches in the validation data loader.
   - Finally, it prints and returns the average validation loss.
4. Regression Model Function (regression_model):
   - This function serves as the main entry point for training and saving the regression model.

- It takes two parameters: data (the path to the data directory) and path (the path to save the trained model).
- The function loads the input data from the instance file and the target labels from the performance file.
- It checks for null values and duplicates in the data to ensure data quality.
- The input data and target labels are converted to tensors (X_data and y_data).
- The dataset is created using the ClassificationDataset class, and the data is split into training and validation sets.
- Data standardization is performed on the training and validation sets using mean and standard deviation.
- Hyperparameters, such as input size, output size, hidden layer size, number of epochs, batch size, learning rate, and weight decay, are set.
- Data loaders are created for the training and validation sets.
- The regression model is instantiated using the Regression class.
- The loss function (MSELoss) and optimizer (Adam) are defined.
- The model is trained for the specified number of epochs, and training loss is printed for each epoch.
- Validation loss is computed by calling the validate function on the validation dataset.
- Early stopping is implemented if the validation loss does not improve for a certain number of epochs.
- Finally, the trained model and related details (input size, output size, hidden layer size, model state_dict, train mean, train std, batch size) are saved to the specified path using torch.save.

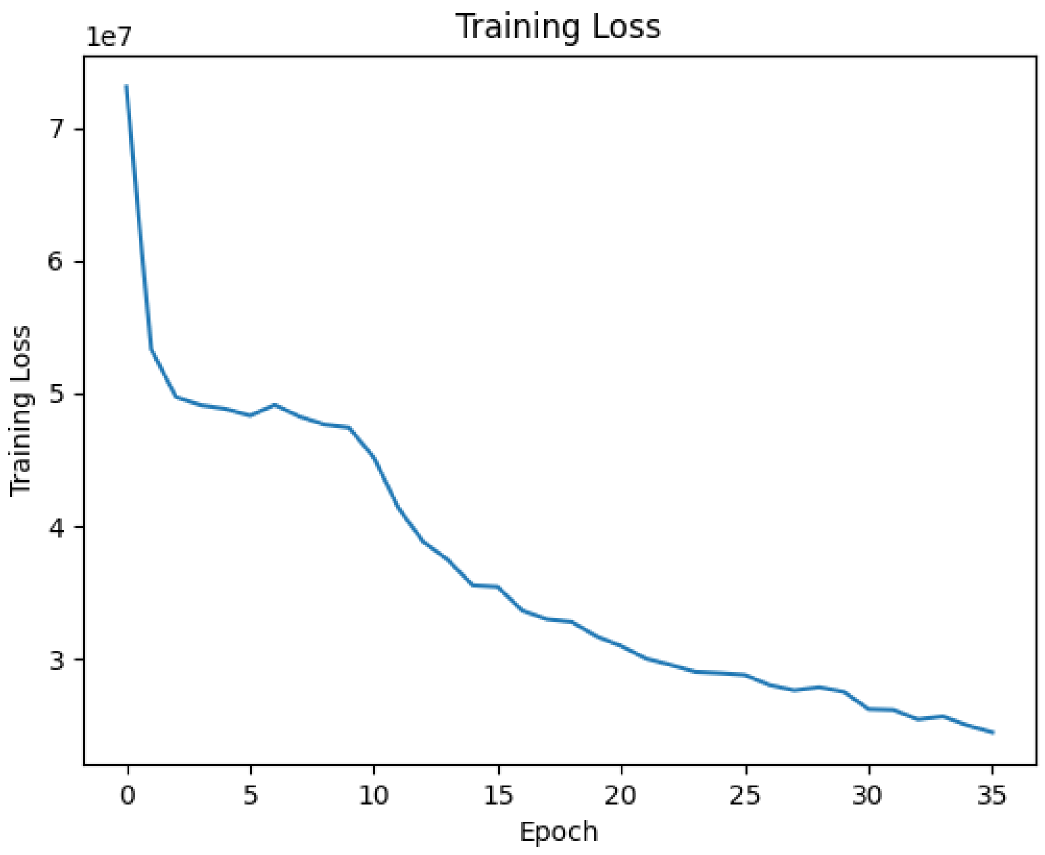

Fig: Fig: Loss vs epoch for regression

e. Classification.py

This code focuses on training a classification model using PyTorch. It shares similarities with the previous code(regression.py) but has a few additional modifications. To summarize the overlapping parts and then explain the additional aspects:

Overlapping Parts:

1. Dataset Class (ClassificationDataset): The ClassificationDataset class is used to create a custom dataset for classification, similar to the previous code. It holds the input data and target algorithm for classification.
2. Validation Function (validate): The validate function is used to evaluate the trained model on the validation dataset, just like in the previous code.
3. Custom Classification Model Class (Classification): The Classification class defines the architecture of the classification model. It consists of input layer, multiple hidden layers, and an output layer with ReLU activation functions in between.

4. Training and Saving the Model: The classification_model function trains the classification model. It loads the input data and target labels, checks for data quality, performs data normalization, creates data loaders, defines the model, loss function, and optimizer, and then trains the model using a loop over epochs. The model is saved after training.

Additional Aspects:

1. Early Stopping: This code introduces early stopping during training. If the validation loss does not improve for a specified number of epochs (controlled by the patience variable), the training is stopped to prevent overfitting.
2. Input Data Format: The input data is read from the "instance-features.txt" file, where each line represents a sample with its corresponding feature values. Similarly, the target labels are read from the "performance-data.txt" file, where each line represents a sample with its corresponding target algorithm.
3. Data Standardization: After splitting the dataset into training and validation sets, the input data is standardized using mean and standard deviation. The same mean and standard deviation values are applied to the validation set as well.
4. Hyperparameters: The hyperparameters, such as hidden layer size, number of epochs, batch size, learning rate, and weight decay, are set explicitly in the code.
5. Model Details: The details of the trained model, including input size, output size, hidden layer size, model state_dict, train mean, train std, and batch size, are saved as a dictionary (model_details) along with the trained model.

In summary, this code extends the previous code by implementing early stopping, adding data standardization, modifying the input data format, and including additional model details in the saved model.

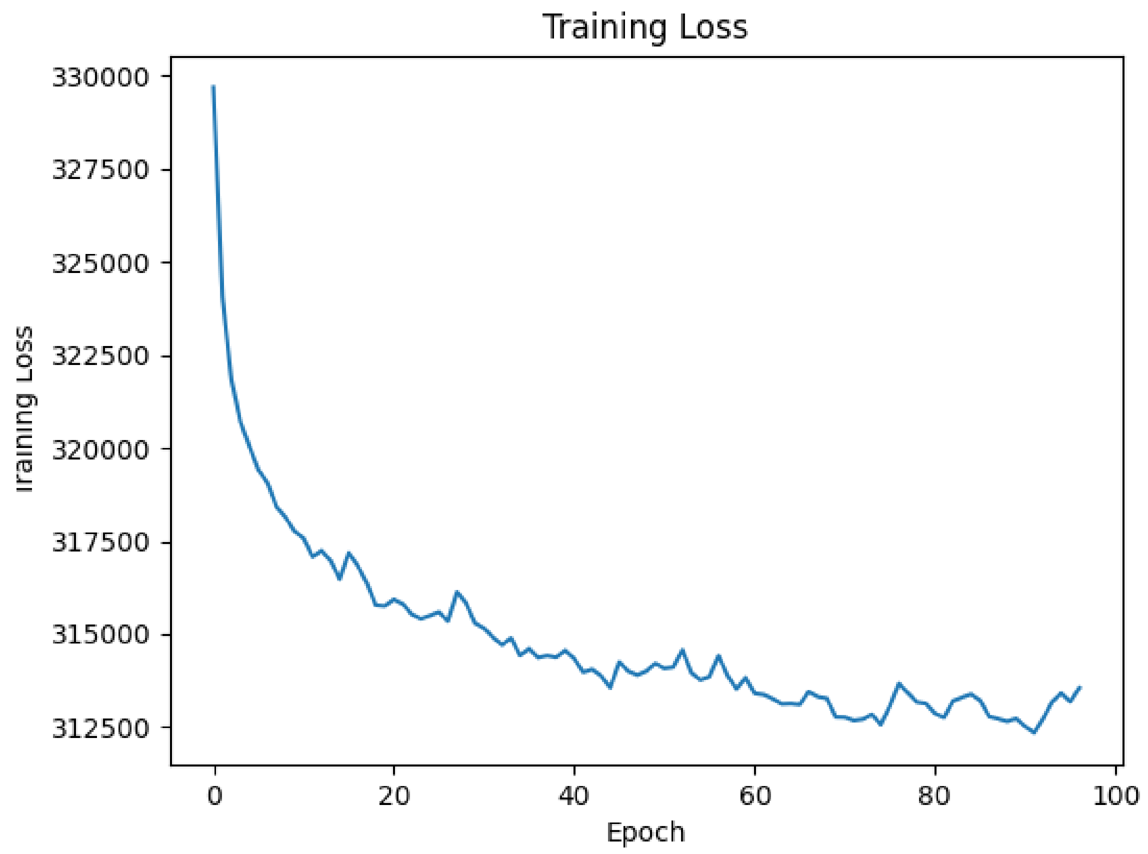

Fig: Loss vs epoch for classification_basic

f. P2_2_cost.py

The provided code implements a classification cost model using PyTorch. Here's a summary of the code and since most of the code is similar except for the loss, an explanation of the custom_loss function:

1. The code defines a custom dataset class called ClassificationDataset inheriting from the Dataset class. It takes input data and target algorithm tensors as arguments. The dataset class implements the necessary methods, such as __len__ and __getitem__, to work with PyTorch's DataLoader for efficient data loading during training.
2. The Classification_cost class is defined as a subclass of nn.Module. It represents the classification cost model and consists of several linear layers with ReLU activations.

3. The validate function is defined to perform validation on the trained model using a validation dataset. It takes the model, validation data loader, and loss criterion as arguments and returns the average validation loss.
4. The custom_loss function calculates a custom loss for the classification problem. Here's a detailed explanation of the steps involved:
    - The function takes the model outputs (outputs) and target algorithm tensor (targets) as inputs.
    - It first obtains the predicted indices with the highest probability using torch.argmax.
    - Next, it extracts the predicted costs corresponding to the predicted indices using torch.gather.
    - Then, it obtains the true indices with the lowest cost values using torch.argmin.
    - The function further extracts the true values corresponding to the true indices using torch.gather.
    - The regret is calculated by subtracting the predicted cost from the true values.
    - The loss function is defined as nn.CrossEntropyLoss with no reduction.
    - The loss is calculated by multiplying the loss from the loss function with the regret (element-wise).
    - Finally, the mean of the calculated loss is returned.
5. The classification_cost_model function is the main function that sets up the model training process. It performs the following steps:
    - Loads the input data and target algorithm data from files.
    - Creates instances of the ClassificationDataset class to hold the data.
    - Splits the dataset into training and validation sets using random_split.
    - Standardizes the data by subtracting the mean and dividing by the standard deviation.
    - Sets hyperparameters such as input/output sizes, hidden layer size, number of epochs, batch size, learning rate, and weight decay.
    - Creates data loaders for the training and validation datasets.

- Creates an instance of the Classification_cost model.
- Defines the optimizer and criterion (custom_loss).
- Trains the model for the specified number of epochs, evaluating the validation loss after each epoch.
- Implements early stopping if the validation loss does not improve for a certain number of epochs.
- Saves the trained model along with necessary details.

To summarize, the custom_loss function calculates a custom loss by comparing the predicted algorithm indices with the true indices, extracting the corresponding costs, and calculating the regret. This loss function is then utilized in the training process of the classification cost model implemented in the code.

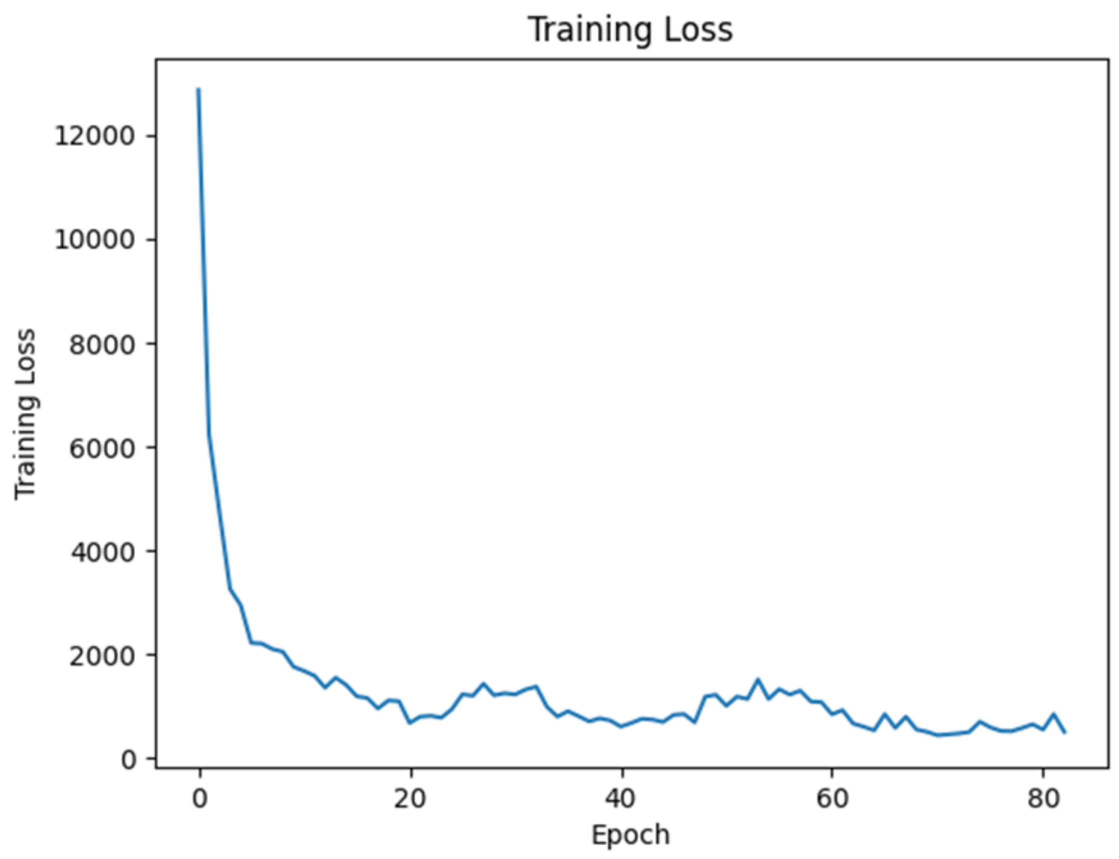

Fig: Loss vs epoch for classification advanced

g. random_forest.py

A simple random forest classifier is implemented using the scikit-learn library for a classification task. Here's a breakdown of what the code does:

1. It imports the necessary libraries: torch for tensor operations, Dataset and random_split from torch.utils.data for dataset handling, PCA from sklearn.decomposition for performing principal component analysis, RandomForestClassifier from sklearn.ensemble for the random forest model, and accuracy_score from sklearn.metrics for evaluating the model's accuracy.

2. It defines a custom dataset class called ClassificationDataset. This class inherits from torch.utils.data.Dataset and takes two inputs: input and algos. In the __init__ method, it initializes these inputs, and in the __len__ and __getitem__ methods, it returns the length of the dataset and retrieves an item at a given index, respectively.

3. It defines the random_forest function that takes two inputs: data (the path to the data) and path (the path to save the model). This function performs the following steps:

- It loads the input data (instance-features.txt) and target data (performance-data.txt) from the given paths. The data is read line by line and converted to tensors using torch.tensor.
- It checks for null values and duplicates in the input and target data tensors.
- It creates a ClassificationDataset object from the input and target data tensors.
- It determines the sizes of the training and validation sets based on a train ratio of 80%.
- It splits the dataset into training and validation sets using random_split from torch.utils.data.
- It normalizes the input data using mean and standard deviation calculated from the training set. This normalization ensures that the data is centered around zero and has unit variance.
- It performs Principal Component Analysis (PCA) on the normalized training data using the PCA class from scikit-learn. PCA is used to reduce the dimensionality of the data.
- It trains a random forest classifier using the RandomForestClassifier class from scikit-learn. The model is trained on the PCA-transformed training data and the corresponding target labels.
- It applies the trained model to predict the target labels for the PCA-transformed validation data.
- It calculates the accuracy of the model by comparing the predicted labels with the actual labels from the validation set using accuracy_score from scikit-learn.

   - It prints the accuracy of the model.

4. There are commented lines at the end of the code for standalone execution. Uncommenting those lines and running the script directly will execute the random_forest function.

Overall, this code demonstrates how to preprocess data, perform PCA dimensionality reduction, and train a random forest classifier for a classification task using scikit-learn.

h. evaluate.py (common)

1. Argument Parsing: The code uses the argparse module to parse command-line arguments. It expects two arguments: --model (path to a trained AS model) and --data (path to a dataset).
2. Model Loading: The trained model is loaded from the specified path using torch.load(). The model's architecture and other details are extracted, including input size, output size, hidden layer size, and training statistics.
3. Data Loading: The test data is loaded from the given dataset directory. The instance features and performance data are read from separate files and converted into tensors.
4. Data Normalization: The loaded data is normalized using the training mean and standard deviation obtained during the model training phase.
5. Dataset and DataLoader: The normalized data is used to create a custom dataset (ClassificationDataset) and a data loader (DataLoader) for efficient batch processing.
6. Evaluation Metrics:
   - Average Loss (avg_loss): The average loss value across the given dataset. It is calculated by summing up the individual losses and dividing by the total number of samples in the dataset. Formula: avg_loss = sum(losses) / num_samples
   - Accuracy (accuracy): Classification accuracy of the model on the dataset. It represents the percentage of correct predictions. Formula: accuracy = (correct_predictions / num_samples) * 100
   - Average Cost (avg_cost): The average cost of the predicted algorithms on the given dataset. It is calculated by taking the mean of the costs for all samples. Formula: avg_cost = sum(costs) / num_samples
   - SBS-VBS Gap (sbs_vbs_gap): The SBS-VBS gap of the model on the given dataset. It measures the difference between the average cost of the SBS and

the average cost of the VBS. Formula: sbs_vbs_gap = (avg_cost - vbs_avg_cost) / (sbs_avg_cost - vbs_avg_cost)

  - SBS Average Cost (sbs_avg_cost): The average cost of the SBS on the given dataset. It represents the minimum cost achieved by any algorithm for each instance. Formula: sbs_avg_cost = mean(min_costs)
  - VBS Average Cost (vbs_avg_cost): The average cost of the VBS on the given dataset. It represents the minimum cost achieved for each instance across all algorithms. Formula: vbs_avg_cost = mean(min_costs_per_instance)

7. Model Evaluation: The code iterates through the data loader, passing input samples through the model and comparing the predictions with the target values. It calculates the average loss, accuracy, and average cost of the predicted algorithms.
8. Results Printing: The final evaluation results, including the average loss, accuracy, average cost, SBS cost, VBS cost, and SBS-VBS gap, are printed to the console. Additionally, the results are appended to a text file named "<model_name>_result.txt" in the "../results/" directory.

Overall, this code evaluates a trained AS model by loading it, making predictions on a given dataset, and computing several evaluation metrics to assess the model's performance.

Analysis

Based on the results obtained, let's analyze and compare the performance of the three types of neural networks. We compare various evaluation metrics such as loss, accuracy, average cost, sbs_cost, vbs_cost, and sbs_vbs_gap.

1. **Regression:**

- Loss: 25,975,487.1000
- Accuracy: 0.3337
- Average cost: 6,652.3823
- SBS cost: 8,690.0127
- VBS cost: 4,001.6335
- SBS-VBS gap: 0.5654

2. **Classification_NN:**

- Loss: 229,878,628.0000

- Accuracy: 0.3994
- Average cost: 4,729.2503
- SBS cost: 8,690.0127
- VBS cost: 4,001.6335
- SBS-VBS gap: 0.1552

3. Classification_cost:

- Loss: 1,641.8295
- Accuracy: 0.0151
- Average cost: 14,860.7043
- SBS cost: 8,690.0127
- VBS cost: 4,001.6335
- SBS-VBS gap: 2.3162

Analysis:

1. Regression model: The regression model has a high loss compared to the Classification_cost model but much lower than the Classification_NN. Its accuracy is moderate at 0.3337, and the average cost is also moderate. The SBS-VBS gap is relatively small, indicating that the model is reasonably close to the best algorithm.
2. Classification_NN model: The Classification_NN model has the highest loss by a significant margin, which generally indicates a poorer model fit. However, it also has the highest accuracy of the three models at 0.3994, suggesting it has the best predictive performance. The average cost is the lowest, indicating the best optimization of the cost across the instance set. The SBS-VBS gap is the smallest, suggesting that this model is the closest to the best algorithm.
3. Classification_cost model: The Classification_cost model has the lowest loss of the three, indicating a potentially better model fit. However, its accuracy is significantly lower than the other models, indicating it may not be as effective at predicting the best algorithms. The average cost is the highest, suggesting poorer optimization of the cost metric. The SBS-VBS gap is the largest, indicating a larger gap between the selected algorithm and the best algorithm.

Conclusion

The Regression model provides a balance between loss, accuracy, and cost, but it doesn't excel in any one area. The Classification_NN model has the highest loss but also the highest accuracy and the lowest average cost, suggesting it might be the best model if the objective is to optimize the average cost metric and predictive accuracy. However, the high loss could indicate overfitting or other issues that might impact the model's generalization to new data. The Classification_cost model has the lowest loss but also the lowest accuracy and the highest average cost, suggesting it might not be as effective for this task.

Given the goal is to optimize the average cost metric rather than classification accuracy, the Classification_NN model appears to be the most suitable, despite its high loss. Future work might focus on techniques to reduce the loss of this model without significantly impacting its accuracy or average cost.