Analysis of tabular and image datasets using ML algorithms and Neural Networks.

Professor:
Dr. Lalitha Sankar

Team Members:
Jessica Jones, PhD Data Science student, Jaikrishna Patil, PhD Data Science student
Vivek Sahukar, PhD Data Science student

December 3, 2024

1 Introduction

1.1 Datasets

The UCI Adult Dataset [1] is a multivariate and tabular dataset. There are fourteen features for this dataset that are both categorical and numerical. The value of target variable is 1 if individual has an income above $50K per year and 0 for those who earn less than 50K per year. This dataset does have missing values and will need to be cleaned up a little before use. The Breast Cancer Wisconsin (Diagnostic) dataset is another multivariate and tabular dataset. There are ten primary features each with three different measurements. The predictive goal of the Breast Cancer Wisconsin (Diagnostic) dataset[2] is to determine whether a breast cancer tumor is malignant (1) or benign (0). Comparatively, it was found to be much cleaner dataset. Fashion-MNIST [3] is a numerical dataset consisting of images. Each image is a 28x28 grayscale pixel image consisting of 10 possible labels. The goal is to predict accurately label of each image, i.e. T-shirt, trousers, pullover, dress, coat, sandal, shirt, sneaker, bag, or ankle boot.

1.2 Machine learning models and Neural networks

There are four algorithms that will be used in this project. The first is Logistic regression is a statistical method used for predicting categorical outcomes based on one or more independent variables. Commonly used for binary classification tasks, it models the probability of an event's occurrence by applying a sigmoid function to a linear combination of input features, mapping them to values between 0 and 1 [4]. Regularization techniques, such as L1 and L2, can be integrated into logistic regression to control model complexity and prevent overfitting. The next algorithm is Support Vector Machine (SVM). SVM is a machine learning algorithm used for classification. For binary classification, SVM finds a hyperplane that maximizes the margin between two classes. It is particularly effective with linearly separable data. For non-linearly separable data, SVM utilizes kernel functions to project the data into a higher-dimensional space, enabling linear separation [5]. K-Nearest Neighbors (KNN) with Principal Component Analysis (PCA) is also used in this project. PCA, a technique for dimensionality reduction. It does this by projecting data onto orthogonal axes that capture the most variance, thus revealing the most significant features of the dataset [6]. This process helps mitigate the curse of dimensionality that KNN faces. By reducing the number of features, PCA not only lessens computational complexity but also enhances KNN's ability to identify meaningful nearest neighbors. KNN classifies data points based on the majority class among its k nearest neighbors, determined using a distance metric (most commonly Euclidean) [7]. The last algorithm we will use is Feedforward Neural Network (FNN) for tabular data, and Convolutional Neural Network (CNN) for image data. FNN is a multilayer perceptron. The input layer of the FNN receives the features of the data. In an FNN, the data flows in one direction from the input layer through one or more hidden layers to the output layer. The output layer is where the classification occurs. During the training process, the FNN adjusts the weights and biases of its connections based on the input data and the correct classification labels. The class with the highest probability or confidence is then assigned as the predicted class for the input data [8]. CNN, however, is designed specifically for analyzing and processing image data. CNN uses convolution layers to capture

important patterns and relationships in images. These layers use filters that slide across the image to find specific features and spatial relationships. The pooling layers reduce the size of the image while preserving the most important features [9]. The fully connected layers then classify the relevant features into different categories to classify the images.

1.3 Team contribution

Jessica Jones: Jessica helped create the outline and rough draft for the final paper. She also completed the introduction, references, and other parts of the report as needed. She peer reviewed results and code of other members. Jessica worked on k-NN with PCA with the breast cancer dataset, logistic regression on the UCI adult dataset. She also revised the rough draft and submitted the final project report for the group.
Jaikrishna Patil: Jaikrishna helped with the outline of the rough draft. Peer reviewed results of the code of other members. He also used SVM and KNN with PCA on the UCI adult dataset. Used logistic regression and SVM on the breast cancer data set. He performed CNN on the Fasion0MNIST dataset. He then filled in the rough draft with his code's output.
Vivek Sahukar: Prepared each dataset for analysis by splitting it into test, train, and validation sets. He reviewed all code for accuracy. He also performed FNN for the breast cancer and UCI data set, and SVM and logistic regression for the Fashion-MNIST dataset. He then updated the report by posting his results. All members contributed to posting results and peer reviewing the report for final submission.

2 Results for Breast Cancer dataset

The libraries were imported (detailed later). The dataset was loaded using `sklearn.datasets.breast_cancer()`. For the exploratory data analysis, missing values were checked using `df.isna().sum().sum()` and none were found. We used `df.target.value_counts()` for checking class imbalance. There is not much class imbalance. We used the parameter in the model `'class_weights'='balanced'` but did not make any difference to the metrics so did not use it. Then, we created a histogram of class distribution and made the correlation matrix to study the correlation amongst features. We then split the dataset into train/validation/test sets using `sklearn.model_selection.train_test_split(stratify=y)` to ensure same distribution of classes in the splits. We normalized the datasets using `StandardScaler().fit_transform()` on training set and `StandardScaler.transform()` on validation and test sets. Then the 3 training, validation, test sets were saved to *csv* files using `pandas.to_csv()`. Since, there is class imbalance, we did not use the `accuracy_score` metric but used `roc_auc_score, precision_score, F1-score` on validation sets to choose the best model.

2.1 Logistic Regression

We loaded the saved datasets. We used the model `sklearn.linear_model.LogisticRegression`. For L1 and L2 regularization, we set the parameter `penalty='l1' or 'l2'` respectively in the model. We trained 3 models (without and with L1 and L2 regularization) and compared the metrics. We did hyperparameter tuning using *gridsearch* over values of C. Best Model found was Logistic Regression with L2 Regularization ($C = 0.1$). We wrote a custom function `plot_roc_pr_curves` to calculate the AUROC and AUPR curves with bootstrapped 95% confidence intervals. We then did the predictions on test set using this final model. The final metrics (mentioned above) were 0.99 Fig 7 and we also plotted the confusion matrix Fig 9.

2.2 SVM

We loaded pre-processed saved datasets. We trained 3 SVM models with 3 different kernels: linear, polynomial, and RBF. We found the best model as SVM with RBF Kernel and used this to make predictions on test set, generate metrics Table 1, Table 2, and also generated the confusion matrix Fig 9.

2.3 PCA+KNN

We loaded the pre-processed saved datasets again. First we ran PCA with 2 components and applied KNN with 3 neighbors to test the workflow. Then we created a pipeline using `sklearn.pipeline.Pipeline` that

Model	Accuracy	Precision	Recall	F1-Score	AUROC	AUPR
SVM no kernel	0.98	0.98	0.97	0.98	1.00	1.00
SVM polyomial kernel	0.91	0.92	0.91	0.91	1.00	1.00
SVM rbf kernel	0.99	0.99	0.99	0.99	1.00	1.00

Table 1: Breast Cancer SVM Results on Validation Set
Precision, Recall, F1 Score are weighted average

Model	Accuracy	Precision	Recall	F1-Score	AUROC	AUPR
SVM rbf kernel	0.97	0.97	0.97	0.97	1.00	1.00

Table 2: Breast Cancer SVM Results on Test Set
Precision, Recall, F1 Score are weighted average

first applied PCA and then KNN. For hyperparameter tuning, we defined a parameter grid to search number of components for PCA and number of neighbors for knn, and used `sklearn.model_selection.GridSearchCV` to find the best hyperparameters as Number of Neighbors = 9 for KNN and Number of Components = 20 for PCA. We ran the same pipeline defined above with best hyperparamters, evaluated this final model on test dataset, generated metrics Table 3, and created the confusion matrix Fig 9.

Dataset	Accuracy	Precision	Recall	F1-Score
Validation	0.99	0.99	0.99	0.99
Test	0.97	0.97	0.96	0.96

Table 3: Breast Cancer PCA+kNN Results after hyperparameter tuning
Precision, Recall, F1 Score are weighted average

2.4 Feedforward Neural Network(FNN)

A 2-layer FNN was used along with relu activation in the end of first layer and sigmoid activation in the end of second linear layer. The model was then trained using Adam optimizer (extended version of SGD) and Binary Cross Entropy loss. The optimal model parameters were decided using Hyperparameter tuning. The following parameters were used for tuning:

```
param_grid = {
    'lr': [0.0001, 0.0005, 0.005, 0.001, 0.01],
    'batch_size': [16, 32, 64],
    'epochs': [50, 100]
}
```

The optimal parameters were `lr=0.0005, epochs=50, batch_size=64`. The learning curve and accuracy for epochs can be seen in Fig. 1. Confusion matrix for test set predictions can be seen in Fig. 3. ROC-AUC curve for the same is depicted in Fig. 2.

Figure 1: Learning curve and accuracy during training(Breast Cancer dataset)

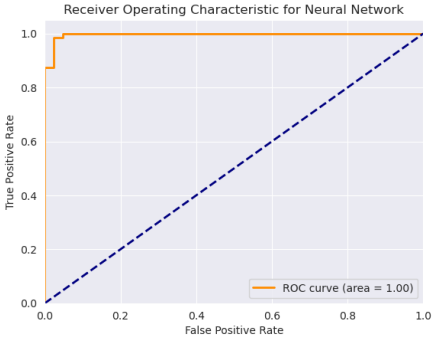


Figure 2: ROC-AUC curve for test set (Breast Cancer dataset)

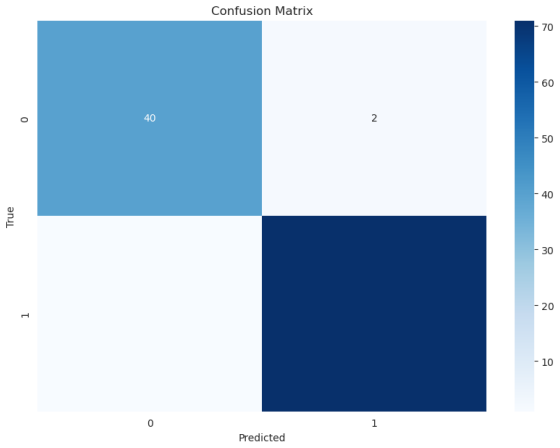


Figure 3: Confusion matrix for test set (Breast Cancer dataset)

3 Results for UCI Adult dataset

The libraries were imported (detailed later). The dataset was loaded using textttsklearn.datasets.fetch_openml(name='adult'). For the exploratory data analysis, missing values were checked using `df.isna().sum().sum()`, the rows having missing values were removed using `df.dropna(subset=['workclass', 'occupation', 'native-country'])` (as instructed). We used `df.target.value_counts()` for checking class imbalance. There is severe class imbalance. We used the parameter in the models `'class_weights'='balanced'` we fit later. The categorical columns were encoded using `sklearn.preprocessing.LabelEncoder`. Doing label encoding made sense for columns: 'workclass', 'education' only, but since there were so many categories, due to which doing One-Hot Encoding would create a wider and sparse dataset, that would be difficult to train. Then, we created a histogram of class distribution and made the correlation matrix for numerical features to study the correlation among features. We then split the dataset into train/validation/test sets using `sklearn.model_selection.train_test_split(stratify=y)` to ensure same distribution of classes in the splits. We normalized the datasets using `StandardScaler().fit_transform()` on training set and `StandardScaler.transform()` on validation and test sets. Then the 3 training, validation, test sets were saved to *csv* files using `pandas.to_csv()`. Since, there is class imbalance, we did not use the `accuracy_score` metric but used `roc_auc_score, precision_score, F1-score` on validation sets to choose the best model.

3.1 Logistic Regression

We loaded the saved datasets. We used the model `sklearn.linear_model.LogisticRegression`. For L1 and L2 regularization, we set the parameter `penalty='l1' or 'l2'` respectively in the model. We trained 3 models (without and with L1 and L2 regularization) and compared the metrics. We did hyperparameter tuning using *gridsearch* over values of C. Best Model found was Logistic Regression with L1 Regularization ($C = 100$). We wrote a custom function `plot_roc_pr_curves` to calculate the AUROC and AUPR curves with bootstrapped 95% confidence intervals. We then did the predictions on test set using this final model. The final metrics (mentioned above): F1-Score was 0.80 Fig 10 and we also plotted the confusion matrix. We calculated the feature importance using the final model and found 'capital-gain', 'education', 'age', 'sex', and 'hours-per-week' were top 5 predictors.

3.2 SVM

We loaded pre-processed saved datasets. We trained 3 SVM models with 3 different kernels: linear, polynomial, and RBF. We found the best model as SVM with RBF Kernel and used this to make predictions on the test set, generate metrics Table 7, Table 9, Fig 11, and also generated the confusion matrix.

Model	Accuracy	Precision	Recall	F1-Score	AUROC	AUPR
SVM linear kernel	0.76	0.81	0.76	0.78	0.85	0.68
SVM polyomial kernel	0.76	0.81	0.76	0.66	0.88	0.71
SVM rbf kernel	0.84	0.84	0.84	0.84	0.90	0.76

Table 4: UCI Adult SVM Results on Validation Set
Precision, Recall, F1 Score are weighted average

3.3 PCA+KNN

We loaded the pre-processed saved datasets again. First we ran PCA with 2 components and applied KNN with 3 neighbors to test the workflow. Then we created a pipeline using `sklearn.pipeline.Pipeline` that first applied PCA and then KNN. For hyperparameter tuning, we defined a parameter grid to search number of components for PCA and number of neighbors for knn, and used *GridSearch* to find the best hyperparameters as Number of Neighbors = 10 for KNN and Number of Components = 10 for PCA. We ran the same pipeline defined above with best hyperparamters, evaluated this final model on test dataset, generated metrics Table 5, and created the confusion matrix.

Dataset	Accuracy	Precision	Recall	F1-Score
Validation	0.80	0.80	0.80	0.75
Test	0.79	0.79	0.79	0.75

Table 5: UCI Adult PCA+kNN Results after hyperparameter tuning
Precision, Recall, F1 Score are weighted average

3.4 FNN

A similar FNN architecture used for breast cancer dataset was used here along with normalization of data. Normalization provided major improvement in the results on the test set for this particular experiment. The optimal parameters given by hyperparamter tuning were `lr=0.001, epochs=15, batch_size=32`. The learning curve and accuracy for epochs can be seen in Fig. 4. Confusion matrix for test set predictions can be seen in Fig. 13. ROC-AUC curve for the same is depicted in Fig. 12.

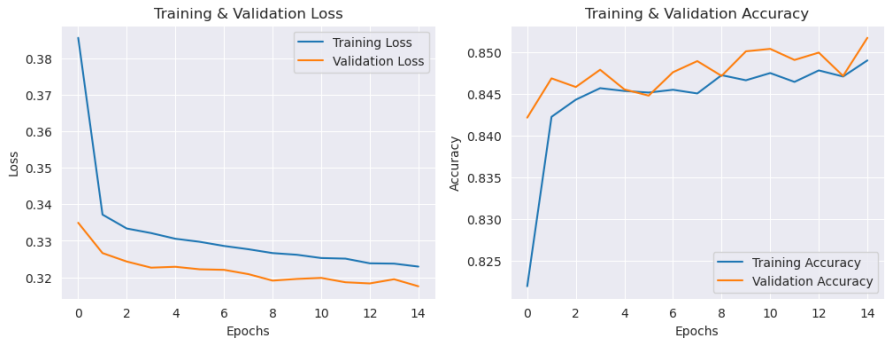


Figure 4: Learning curve and accuracy during training(UCI-Adult dataset)

4 Results for Fashion MNIST dataset

The libraries were imported (detailed later). The dataset was loaded using `torchvision.datasets.FashionMNIST)`. For the exploratory data analysis, we used `np.bincount()` for checking class imbalance and there were same count of each class in train, validation and test sets, so no class imbalance existed. We then split the dataset into train/validation/test sets and saved using `np.save()`. We normalized the datasets using `torchvision.transforms.Compose()` on training set and used that transform to normalize the validation and test sets. We did not use the `accuracy_score` metric but used `roc_auc_score, precision_score, F1-score` on validation sets to choose the best model. **Since this is a very big dataset, we used the NVIDIA Rapids Library [10] that allows to run the whole machine learning model pipeline, viz. data loading, model training, and inference on GPU, which resulted in a speed-up by a significant amount as compared to the scikit-learn library.**

4.1 Logistic Regression

We loaded the saved datasets and converted to `cupy arrays`. We used the model `sklearn.linear_model.LogisticRegression`. For L1 and L2 regularization, we set the parameter `penalty='l1' or 'l2'` respectively in the model. We trained 3 models (without and with L1 and L2 regularization) and compared the metrics. We did hyperparameter tuning using *gridsearch* over values of C. Best Model found was Logistic Regression with L1 Regularization ($C = 0.01$). We then did the predictions on test set using this final model. The final metrics (mentioned above) Table 6 were 0.99.

Dataset	Accuracy	Precision	Recall	F1-Score
Validation	0.86	0.86	0.86	0.86
Test	0.85	0.84	0.84	0.84

Table 6: Fashion MNIST Logistic Regression with L1 Regularization after hyperparameter tuning C=0.01 Precision, Recall, F1 Score are weighted average

4.2 SVM

We loaded pre-processed saved datasets. We trained 3 SVM models with 3 different kernels: linear, polynomial, and RBF. We found the best model as SVM with RBF Kernel and used this to make predictions on the test set, generate metrics Table 7, Table 10, and also generated the confusion matrix.

4.3 PCA+KNN

We loaded the pre-processed saved datasets again. First we ran PCA with 2 components and applied KNN with 3 neighbors to test the workflow. Then we created a pipeline using `sklearn.pipeline.Pipeline` that

Model	Accuracy	Precision	Recall	F1-Score
SVM linear kernel	0.86	0.86	0.86	0.86
SVM polyomial kernel	0.86	0.88	0.88	0.88
SVM rbf kernel	0.90	0.90	0.90	0.90

Table 7: Fashion MNIST SVM Results on Validation Set
Precision, Recall, F1 Score are weighted average

first applied PCA and then KNN. For hyperparameter tuning, we defined a parameter grid to search number of components for PCA and number of neighbors for knn, and used `sklearn.model_selection.GridSearchCV` to find the best hyperparameters as Number of Neighbors = 10 for KNN and Number of Components = 100 for PCA. We ran the same pipeline defined above with best hyperparamters, evaluated this final model on test dataset, generated metrics Table 8, and created the confusion matrix.

Dataset	Accuracy	Precision	Recall	F1-Score
Validation	0.87	0.87	0.87	0.87
Test	0.86	0.86	0.86	0.86

Table 8: Fashion MNIST PCA+KNN after hyperparameter tuning
Precision, Recall, F1 Score are weighted average

4.4 CNN

Both 1-layer CNN and 2-layer CNN with kernel of `kernel size = 3, padding=1, stride=1` was used to compare the results from both of the architectures . It was found that 2-layer CNN gave slightly better results (Accuracy on test set: 91.70%) than the 1-layer CNN(Accuracy on test set: 90.45%). The optimal parameters given by hyperparamter tuning were `lr=0.001, epochs=100, batch_size=64`. The learning curve and accuracy for epochs can be seen in Fig. 5. Confusion matrix for test set predictions can be seen in Fig. 6.

Figure 5: Learning curve and accuracy during training - 2 layer CNN (Fashion MNIST dataset)

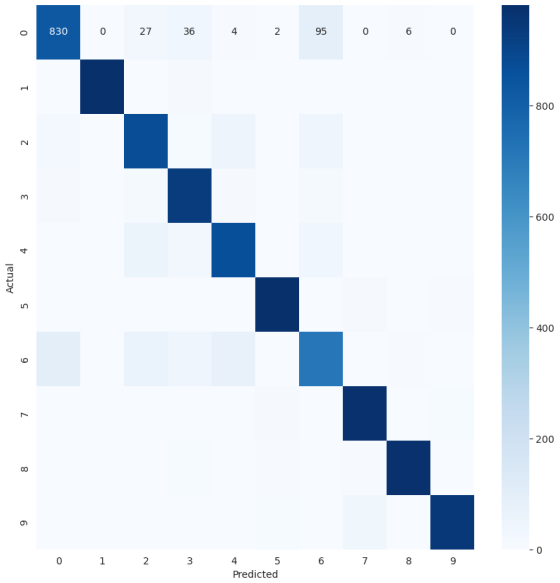


Figure 6: Confusion matrix for test set - 2 layer CNN (Fashion MNIST dataset)

5 About learned concepts

This project encompasses a diverse amount of knowledge about machine learning. Many of the topics taught in EEE549 were crucial to understanding and completing the project. From understanding the basics, such as the overall ML pipeline, to the advanced neural networks applications, this project contained almost all the topics covered in the course. One of the key takeaways from this project was understanding and using machine learning models on multiple types of datasets. Each model has its own pros and cons, and each dataset may work better for a different model. For each model, we had to understand how to optimize using validation data, and loss functions. This was a key idea taught throughout the course. Another key idea in multiple models is the sigmoid function $\sigma = \frac{1}{1+\exp(-z)}$. It shows up in both logistic [4] and neural networks [9]. The previous assignments in the course also helped prepare students to perform k-fold cross validation, hyperparameter tuning, data visualization and analysis that were all key components of this project. The project also dived into deep learning using neural networks. This includes learning about low-dimension representations of neural networks, which is similar to feature selection used in traditional machine learning. This can be done in neural networks by pooling. Pooling reduces the spatial space (downsizes) the image to capture key features [9].

6 Conclusion

For the Wisconsin Breast Cancer Data Set, logistic regression was the best algorithm giving accuracy of 98% on test set. Logistic regression is a simple and interpretive algorithm, making it easy to understand and implement. It works well with binary classification problems, where the target variable has two classes, making it a good candidate for the Wisconsin Breast Cancer dataset. Logistic regression is not good for all scenarios, it can struggle when there are too many irrelevant features present and does not work as well for more complex datasets, like FashionMNIST and UCI Adult. After running all the algorithms on the Wisconsin Breast Cancer dataset, we concluded that Logistic Regression with regularization yielded the best results. Comparing the results from Figure 9a, Logistic regression only made one false negative prediction and zero false positive predictions. For the UCI-Adult dataset we found both SVM and KNN provided good results, however KNN was still the best match. SVM is an effective algorithm in high-dimensional space that can handle complex non-linear problems using different kernel functions. It is computationally expensive, however, with large data sets, hand finding the right kernel function can be challenging. This algorithm could be a good match for a complex dataset with a several features like UCI-Adult. There was a better fit however, K-Nearest Neighbors (KNN) can handle non-linear and complex relationships. It is effective when there is a large amount of data available, like the UCI-Adult dataset with 48842 instances [1]. It can be computationally expensive, but PCA can help ease that expense by reducing the irrelevant features. Although Feed Forward Neural Networks (FNN) has many pros, and does well overall on many different tabular datasets, it was not the best choice for the two tabular datasets. FNN can be computationally expensive, and since the results of FNN did not do significantly better than the previous algorithms mentioned, it was not the best fit for either tabular datasets for this project. The final dataset we worked on was an image dataset FasionMNIST. This dataset was large and complex. It was computationally expensive to work on, and many of the previous models did not do a good job predicting the correct labels. Aside from being the only image dataset, it was also the only multiclass dataset. For this reason, the Convolution Neural Network (CNN) was the best algorithm for this function. CNNs are computationally expensive and require a lot of training data to perform well. FasionMNIST is large enough to be a good candidate for CNN. CNN was the best model for FasionMNIST with the higher accuracy rate, see Figure 5.

7 Libraries

We used the NVIDIA Rapids library to run the whole machine learning model pipeline on GPU. The documentation can be found here. Please refer to the installation guide here.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_breast_cancer, fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, auc, make_scorer, roc_auc_score,
roc_curve, average_precision_score, precision_recall_curve,
classification_report, confusion_matrix
from sklearn.utils import resample
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import GridSearchCV

from sklearn.utils import resample
import cudf
import cupy as cp
import cuml
from cuml.svm import SVC, LinearSVC
```

References

[1] B. Becker and R. Kohavi, "Adult." UCI Machine Learning Repository, 1996. DOI: https://doi.org/10.24432/C5XW20.

[2] M. O. S. N. Wolberg, William and W. Street, "Breast Cancer Wisconsin (Diagnostic)." UCI Machine Learning Repository, 1995. DOI: https://doi.org/10.24432/C5DW2B.

[3] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," 2017.

[4] L. Sankar, "Classification logistic regression." EEE549 Lecture, 2023. Arizona State University.

[5] L. Sankar, "Svm-ridge-to-kernels-part1." EEE549 Lecture, 2023. Arizona State University.

[6] L. Sankar, "Principal component analysis ref: Hastie et al., ch. 14." EEE549 Lecture, 2023. Arizona State University.

[7] L. Sankar, "Nearest neighbor methods." EEE549 Lecture, 2023. Arizona State University.

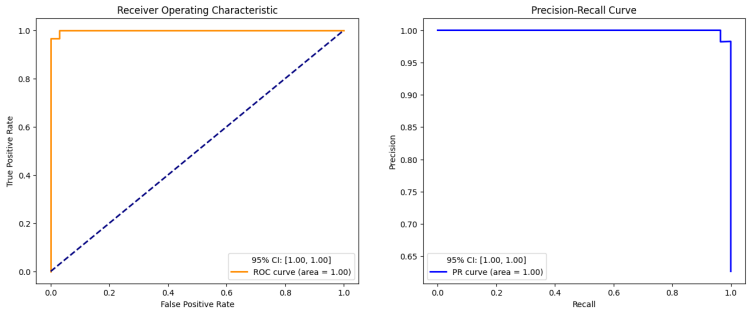
[8] G. T. Network, "Galaxy training: Feedforward neural networks (fnn) deep learning - part 1," 2023. DOI: https://training.galaxyproject.org/training-material/topics/statistics/tutorials/FNN/slides-plain.html.

[9] L. Sankar, "Brief introduction to neural networks." EEE549 Lecture, 2023. Arizona State University.

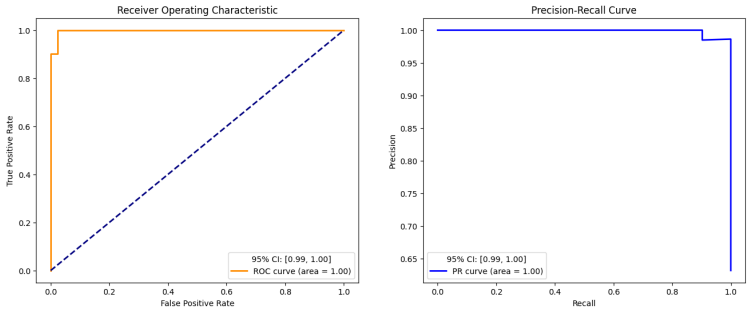
[10] "RAPIDS: Open source gpu data science." https://rapids.ai/. Accessed: [Nov 29, 2023].

8 Appendix

8.1 Plots



(a) Validation set



(b) Test set

Figure 7: AUROC, PR curves for Logistic Regression with L2 regularization after hyperparameter tuning

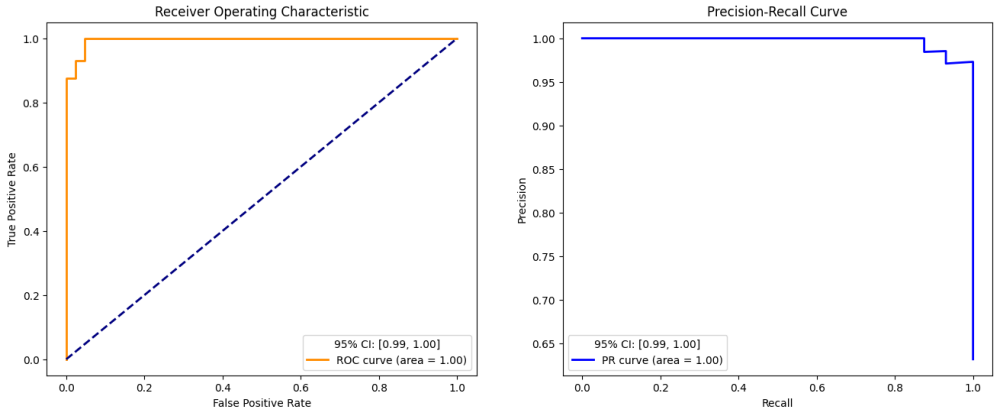
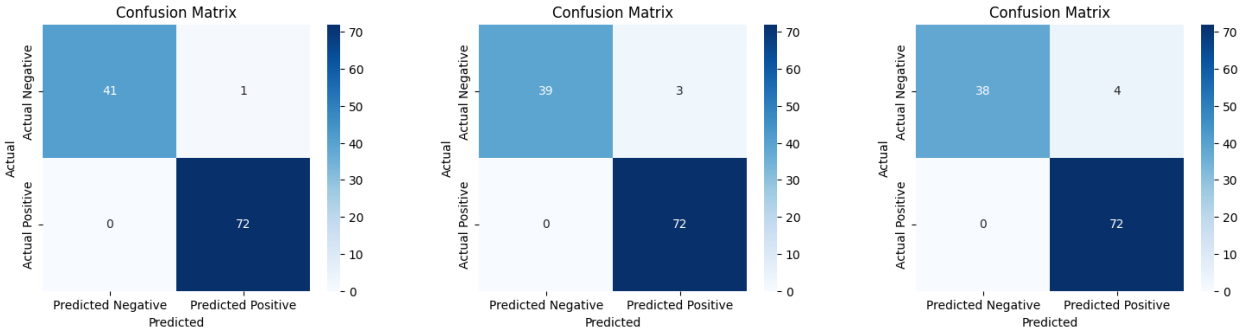


Figure 8: AUROC and PR curves for test set with bootstrapped confidence intervals for Support Vector Classifier model

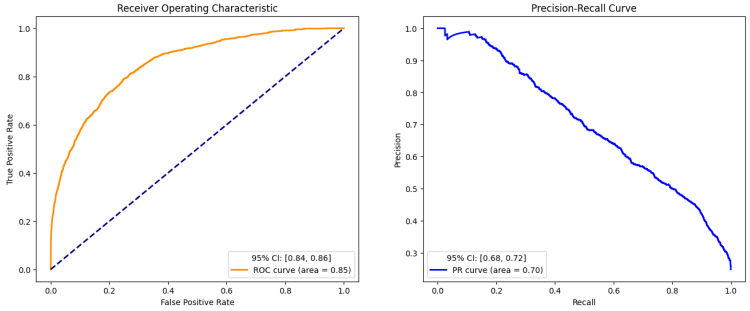


(a) Logistic regression model with L2 regularization after hyperparameter tuning

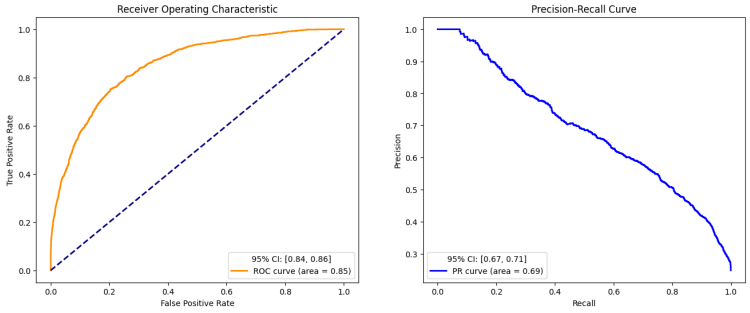
(b) Support Vector Classifier model

(c) PCA-KNN model

Figure 9: Confusion matrix for Test Set for Breast Cancer Dataset



(a) Validation set



(b) Test set

Figure 10: AUROC, PR curves - Logistic Regression L1 regularization after hyperparameter tuning C=0.1

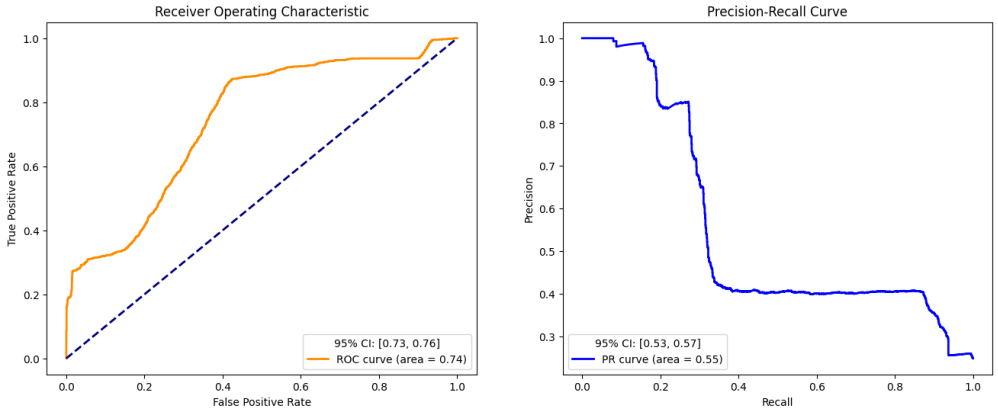


Figure 11: AUROC and PR curves for test set with bootstrapped confidence intervals for Support Vector Classifier - no kernel - model

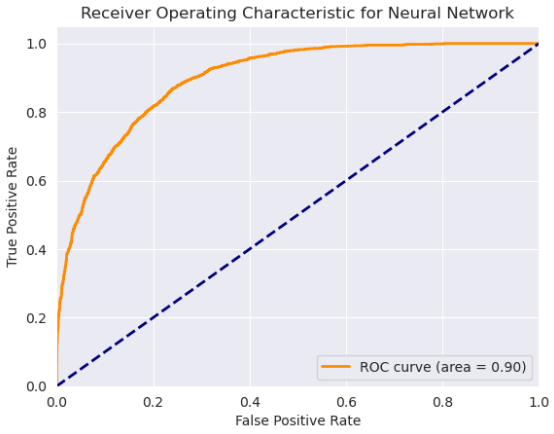


Figure 12: ROC-AUC curve for test set (UCI-Adult dataset)

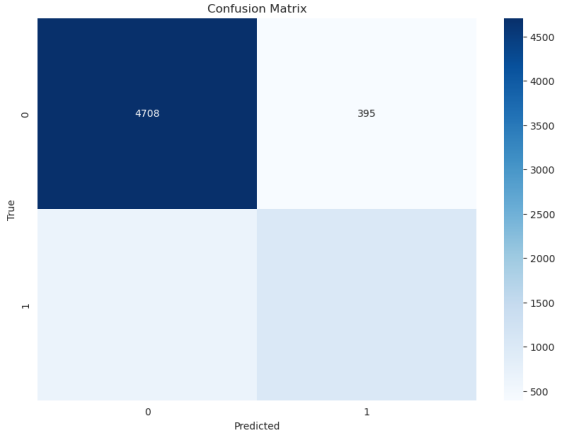


Figure 13: Confusion matrix for test set (UCI-Adult dataset)

8.2 Extra Tables

Model	Accuracy	Precision	Recall	F1-Score	AUROC	AUPR
SVM rbf kernel	0.83	0.83	0.83	0.83	0.89	0.73
SVM linear kernel	0.77	0.81	0.77	0.78	0.85	0.68

Table 9: UCI Adult SVM Results on Test Set
Precision, Recall, F1 Score are weighted average

Model	Accuracy	Precision	Recall	F1-Score
SVM rbf kernel	0.90	0.88	0.88	0.88

Table 10: Fashion MNIST SVM Results of Best Model on Test Set
Precision, Recall, F1 Score are weighted average

8.3 How to run the code

8.3.1 Requirements.txt file to install libraries

```
actionlib==1.14.0
aiohttp==3.9.0
aiosignal==1.3.1
angles==1.9.13
anyio==4.1.0
argon2-cffi==23.1.0
argon2-cffi-bindings==21.2.0
arrow==1.3.0
asttokens==2.4.1
async-timeout==4.0.3
attrs==23.1.0
beautifulsoup4==4.12.2
bleach==6.1.0
bokeh==3.3.1
bondpy==1.8.6
cachetools==5.3.2
camera-calibration==1.17.0
camera-calibration-parsers==1.12.0
catkin==0.8.10
certifi==2023.11.17
cffi==1.16.0
charset-normalizer==3.3.2
click==8.1.7
click-plugins==1.1.1
cligj==0.7.2
cloudpickle==3.0.0
colorcet==3.0.1
comm==0.2.0
contourpy==1.2.0
controller-manager==0.20.0
controller-manager-msgs==0.20.0
cucim==23.10.0
cuda-python==12.3.0
cudf-cu12==23.10.2
cugraph-cu12==23.10.0
cuml-cu12==23.10.0
cuproj-cu12==23.10.0
cupy-cuda12x==12.2.0
cuspatial-cu12==23.10.0
cuxfilter-cu12==23.10.0
cv-bridge==1.16.2
cycler==0.12.1
daal==2024.0.0
daal4py==2024.0.0
dask==2023.9.2
dask-cuda==23.10.0
dask-cudf-cu12==23.10.2
datashader==0.16.0
debugpy==1.8.0
decorator==5.1.1
```

```
defusedxml==0.7.1
diagnostic-analysis==1.11.0
diagnostic-common-diagnostics==1.11.0
diagnostic-updater==1.11.0
distributed==2023.9.2
dynamic-reconfigure==1.7.3
exceptiongroup==1.2.0
executing==2.0.1
fastjsonschema==2.19.0
fastrlock==0.8.2
fiona==1.9.5
flexbe-core==1.4.0
flexbe-input==1.4.0
flexbe-mirror==1.4.0
flexbe-onboard==1.4.0
flexbe-states==1.4.0
flexbe-testing==1.4.0
flexbe-widget==1.4.0
fonttools==4.45.1
fqdn==1.5.1
frozenlist==1.4.0
fsspec==2023.10.0
gazebo_plugins==2.9.2
gazebo_ros==2.9.2
gencpp==0.7.0
geneus==3.0.0
genlisp==0.4.18
genmsg==0.6.0
gennodejs==2.0.2
genpy==0.6.15
geopandas==0.14.1
holoviews==1.18.1
idna==3.6
image-geometry==1.16.2
importlib-metadata==6.8.0
interactive-markers==1.12.0
ipykernel==6.26.0
ipython==8.18.0
isoduration==20.11.0
jedi==0.19.1
Jinja2==3.1.2
joblib==1.3.2
joint-state-publisher==1.15.1
joint-state-publisher-gui==1.15.1
jsonpointer==2.4
jsonschema==4.20.0
jsonschema-specifications==2023.11.1
jupyter-events==0.9.0
jupyter_client==8.6.0
jupyter_core==5.5.0
jupyter_server==2.10.1
jupyter_server_proxy==4.1.0
jupyter_server_terminals==0.4.4
jupyterlab_pygments==0.3.0
```

```
kiwisolver==1.4.5
laser_geometry==1.6.7
lazy_loader==0.3
linkify-it-py==2.0.2
llvmlite==0.40.1
locket==1.0.0
Markdown==3.5.1
markdown-it-py==3.0.0
MarkupSafe==2.1.3
matplotlib==3.8.2
matplotlib-inline==0.1.6
mdit-py-plugins==0.4.0
mdurl==0.1.2
message-filters==1.16.0
mistune==3.0.2
moveit-commander==1.1.13
moveit-core==1.1.13
moveit-ros-planning-interface==1.1.13
moveit-ros-visualization==1.1.13
msgpack==1.0.7
multidict==6.0.4
multipledispatch==1.0.0
nbclient==0.9.0
nbconvert==7.11.0
nbformat==5.9.2
nest-asyncio==1.5.8
numba==0.57.1
numpy==1.24.4
nvtx==0.2.8
overrides==7.4.0
packaging==23.2
pandas==1.5.3
pandocfilters==1.5.0
panel==1.3.2
param==2.0.1
parso==0.8.3
partd==1.4.1
pexpect==4.9.0
Pillow==10.1.0
platformdirs==4.0.0
prometheus-client==0.19.0
prompt-toolkit==3.0.41
protobuf==4.25.1
psutil==5.9.6
ptyprocess==0.7.0
pure-eval==0.2.2
pyarrow==12.0.1
pycparser==2.21
pyct==0.5.0
Pygments==2.17.2
pylibcugraph-cu12==23.10.0
pylibraft-cu12==23.10.0
pynvml==11.4.1
pyparsing==3.1.1
```

```
pyproj==3.6.1
python-dateutil==2.8.2
python-json-logger==2.0.7
python-qt-binding==0.4.4
pytz==2023.3.post1
pyviz_comms==3.0.0
PyYAML==6.0.1
pyzmq==25.1.1
qt-dotgraph==0.4.2
qt-gui==0.4.2
qt-gui-cpp==0.4.2
qt-gui-py-common==0.4.2
raft-dask-cu12==23.10.0
referencing==0.31.0
requests==2.31.0
resource_retriever==1.12.7
rfc3339-validator==0.1.4
rfc3986-validator==0.1.1
rich==13.7.0
rmm-cu12==23.10.0
rosbag==1.16.0
rosboost-cfg==1.15.8
rosclean==1.15.8
roscreate==1.15.8
rosgraph==1.16.0
roslaunch==1.16.0
roslib==1.15.8
roslint==0.12.0
roslz4==1.16.0
rosmake==1.15.8
rosmaster==1.16.0
rosmsg==1.16.0
rosnode==1.16.0
rosparam==1.16.0
rospy==1.16.0
rosservice==1.16.0
rostest==1.16.0
rostopic==1.16.0
rosunit==1.15.8
roswtf==1.16.0
rpds-py==0.13.1
rqt-console==0.4.12
rqt-image-view==0.4.17
rqt-logger-level==0.4.12
rqt-moveit==0.5.11
rqt-reconfigure==0.5.5
rqt-robot-dashboard==0.5.8
rqt-robot-monitor==0.5.15
rqt-runtime-monitor==0.5.10
rqt-rviz==0.7.0
rqt-tf-tree==0.6.4
rqt_action==0.4.9
rqt_bag==0.5.1
rqt_bag_plugins==0.5.1
```

```
rqt_dep==0.4.12
rqt_graph==0.4.14
rqt_gui==0.5.3
rqt_gui_py==0.5.3
rqt_launch==0.4.9
rqt_msg==0.4.10
rqt_nav_view==0.5.7
rqt_plot==0.4.13
rqt_pose_view==0.5.11
rqt_publisher==0.4.10
rqt_py_common==0.5.3
rqt_py_console==0.4.10
rqt_robot_steering==0.5.12
rqt_service_caller==0.4.10
rqt_shell==0.4.11
rqt_srv==0.4.9
rqt_top==0.4.10
rqt_topic==0.4.13
rqt_web==0.4.10
rviz==1.14.20
scikit-learn==1.3.2
scikit-learn-intelex==2024.0.0
scipy==1.11.4
seaborn==0.13.0
Send2Trash==1.8.2
sensor-msgs==1.13.1
shapely==2.0.2
simpervisor==1.0.0
six==1.16.0
smach==2.5.2
smach-ros==2.5.2
smclib==1.8.6
sniffio==1.3.0
sortedcontainers==2.4.0
soupsieve==2.5
srdfdom==0.6.4
stack-data==0.6.3
tbb==2021.11.0
tblib==3.0.0
terminado==0.18.0
tf==1.13.2
tf-conversions==1.13.2
tf2-geometry-msgs==0.7.7
tf2-kdl==0.7.7
tf2-py==0.7.7
tf2-ros==0.7.7
threadpoolctl==3.2.0
tinycss2==1.2.1
toolz==0.12.0
topic-tools==1.16.0
tornado==6.3.3
tqdm==4.66.1
traitlets==5.13.0
treelite==3.9.1
```

```
treelite-runtime==3.9.1
types-python-dateutil==2.8.19.14
typing_extensions==4.8.0
uc-micro-py==1.0.2
ucx-py-cu12==0.34.0
urdfdom-py==0.4.6
uri-template==1.3.0
urllib3==2.1.0
wcwidth==0.2.12
webcolors==1.13
webencodings==0.5.1
websocket-client==1.6.4
xacro==1.14.16
xarray==2023.11.0
xyzservices==2023.10.1
yarl==1.9.3
zict==3.0.0
zipp==3.17.0
```

Unzip the folder, and run the Jupyter notebooks where they are, they are set up to load libraries and datasets according to the folder structure.

We used the NVIDIA Rapids library to run the whole machine learning model pipeline on GPU. The documentation can be found here. Please refer to the installation guide here.

These are all the library imports we did in our code.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_breast_cancer, fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, auc, make_scorer, roc_auc_score,
roc_curve, average_precision_score, precision_recall_curve,
classification_report, confusion_matrix
from sklearn.utils import resample
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import GridSearchCV

from sklearn.utils import resample
import cudf
import cupy as cp
import cuml
from cuml.svm import SVC, LinearSVC
```

2.1 Breast Cancer Logistic Regression Code

```
import sys
sys.path.append('../')
from imports import *
from sklearn.utils import resample

# Loading the datasets
X_train = pd.read_csv('./breast-cancer-processed-data/X_train.csv')
y_train = pd.read_csv('./breast-cancer-processed-data/y_train.csv')
X_val = pd.read_csv('./breast-cancer-processed-data/X_val.csv')
y_val = pd.read_csv('./breast-cancer-processed-data/y_val.csv')
X_test = pd.read_csv('./breast-cancer-processed-data/X_test.csv')
y_test = pd.read_csv('./breast-cancer-processed-data/y_test.csv')

y_train = y_train.squeeze()
y_val = y_val.squeeze()
y_test = y_test.squeeze()

def plot_roc_pr_curves(y_true, y_probs, n_bootstraps=1000):
    """Perform bootstrapping to calculate 95% confidence intervals for
    ROC and PR curves and plot them"""
    bootstrap_auroc_scores = []
    bootstrap_average_precision_scores = []

    for _ in range(n_bootstraps):
        # Bootstrap sample (with replacement)
        indices = resample(np.arange(len(y_true)), replace=True)
        y_true_boot = y_true[indices]
        y_probs_boot = y_probs[indices]

        # Compute metrics for bootstrap sample
        bootstrap_auroc_scores.append(roc_auc_score(y_true_boot, y_probs_boot))
        bootstrap_average_precision_scores.append(average_precision_score(
        y_true_boot, y_probs_boot))

    # Compute confidence intervals
    auroc_lower = np.percentile(bootstrap_auroc_scores, 2.5)
    auroc_upper = np.percentile(bootstrap_auroc_scores, 97.5)
    ap_lower = np.percentile(bootstrap_average_precision_scores, 2.5)
    ap_upper = np.percentile(bootstrap_average_precision_scores, 97.5)

    # Calculate original ROC and PR curves
    fpr, tpr, _ = roc_curve(y_true, y_probs)
    precision, recall, _ = precision_recall_curve(y_true, y_probs)
    auroc = roc_auc_score(y_true, y_probs)
    average_precision = average_precision_score(y_true, y_probs)

    # Plotting
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

    # ROC Curve
    ax1.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {auroc:.2f})')
    ax1.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
```

```
    ax1.set_xlabel('False Positive Rate')
    ax1.set_ylabel('True Positive Rate')
    ax1.set_title('Receiver Operating Characteristic')
    ax1.legend(loc="lower right", title=f'95% CI: [{auroc_lower:.2f}, {auroc_upper:.2f}]')

    # Precision-Recall Curve
    ax2.plot(recall, precision, color='blue', lw=2, label=f'PR curve
    (area = {average_precision:.2f})')
    ax2.set_xlabel('Recall')
    ax2.set_ylabel('Precision')
    ax2.set_title('Precision-Recall Curve')
    ax2.legend(loc="lower left", title=f'95% CI: [{ap_lower:.2f}, {ap_upper:.2f}]')

    plt.show()

    # Return the confidence intervals
    return (np.round(auroc_lower,2), np.round(auroc_upper,2)),
    (np.round(ap_lower,2), np.round(ap_upper,2))

# Initialize the Logistic Regression model without regularization
log_reg = LogisticRegression(penalty=None, solver='lbfgs', max_iter=1000, random_state=1)

# Train the model on the training set
log_reg.fit(X_train, y_train)

# Predict on the validation set
y_val_pred = log_reg.predict(X_val)

# Evaluate the model on the validation set
val_accuracy = accuracy_score(y_val, y_val_pred)
print(f"Validation Set Accuracy: {val_accuracy:.4f}")

# Predict probabilities
y_val_probs = log_reg.predict_proba(X_val)[:, 1]

# Compute AUROC
auroc = roc_auc_score(y_val, y_val_probs)
print(f"Area under the ROC curve: {auroc:.2f}")

# Compute Precision-Recall curve and its area
average_precision = average_precision_score(y_val, y_val_probs)
print(f"Area under the Precision-Recall curve: {average_precision:.2f}")

# AUROC and PR curves with bootstrapped 95% confidence intervals
# for logistic regression with no regularization
auroc_confidence_interval, ap_confidence_interval =
plot_roc_pr_curves(y_val, y_val_probs)

# Initialize the Logistic Regression model with L1 regularization
log_reg_l1 = LogisticRegression(penalty='l1', C=1.0,
solver='liblinear', max_iter=1000, random_state=1)

# Train the model on the training set
log_reg_l1.fit(X_train, y_train)
```

```
# Predict probabilities for the validation and test sets
y_val_probs_l1 = log_reg_l1.predict_proba(X_val)[:, 1]
y_test_probs_l1 = log_reg_l1.predict_proba(X_test)[:, 1]

# Compute the AUROC and average precision score on the validation set
val_auroc = roc_auc_score(y_val, y_val_probs_l1)
val_average_precision = average_precision_score(y_val, y_val_probs_l1)

# Compute the AUROC and average precision score on the test set
test_auroc = roc_auc_score(y_test, y_test_probs_l1)
test_average_precision = average_precision_score(y_test, y_test_probs_l1)

# AUROC and PR curves with bootstrapped 95% confidence intervals
# for logistic regression with L1 regularization
auroc_confidence_interval, ap_confidence_interval =
plot_roc_pr_curves(y_val, y_val_probs_l1)

# Define a set of C values to try
C_values = [0.001, 0.01, 0.1, 1, 10, 100]

# Initialize variables to store the best score and corresponding C value
best_score = 0
best_C = None

# Perform grid search over the C values
for C in C_values:
    # Initialize and train the Logistic Regression model with L1 regularization
    log_reg_l1 = LogisticRegression(penalty='l1', C=C, solver='liblinear',
    max_iter=1000, random_state=1)
    log_reg_l1.fit(X_train, y_train)

    # Evaluate on the validation set
    y_val_probs = log_reg_l1.predict_proba(X_val)[:, 1]
    score = roc_auc_score(y_val, y_val_probs)

    # If the score is better than the best score, update the best score and best C
    if score > best_score:
        best_score = score
        best_C = C

# Output the best C value
print(f"Best C value: {best_C} with AUROC: {best_score:.4f}")

# Train a new model using the best C value
best_log_reg_l1 = LogisticRegression(penalty='l1', C=best_C,
solver='liblinear', max_iter=1000, random_state=1)
best_log_reg_l1.fit(X_train, y_train)

# Continue with bootstrapping, plotting, and evaluation using the best_log_reg_l1 model

# Predict probabilities for the validation and test sets
y_val_probs_l1 = best_log_reg_l1.predict_proba(X_val)[:, 1]
```

```
# AUROC and PR curves with bootstrapped 95% confidence intervals for
# logistic regression with L1 regularization
auroc_confidence_interval_l1, ap_confidence_interval_l1 =
plot_roc_pr_curves(y_val, y_val_probs_l1)

# Define a logistic regression model. Note: The default penalty is 'l2'.
log_reg_l2 = LogisticRegression(solver='liblinear', random_state=1)

# Define the hyperparameter grid for 'C'
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

# Define the scoring function you want to optimize for during the grid search
scorer = make_scorer(roc_auc_score, needs_proba=True)

# Set up the grid search with cross-validation
grid_search = GridSearchCV(log_reg_l2, param_grid, cv=5, scoring=scorer)

# Perform grid search on the training set
grid_search.fit(X_train, y_train)

# Output the best C value
print(f"Best parameters: {grid_search.best_params_}")

# After grid search, the best hyperparameter value is stored in 'best_estimator_'
best_log_reg_l2 = grid_search.best_estimator_

# Now using 'best_l2_model' to make predictions and evaluate it
y_val_probs_l2 = best_log_reg_l2.predict_proba(X_val)[:, 1]

# Evaluate the model using the validation  sets
val_auroc_l2 = roc_auc_score(y_val, y_val_probs_l2)

# AUROC and PR curves with bootstrapped 95% confidence intervals
# for logistic regression with L2 regularization after hyperparameter tuning
auroc_confidence_interval_l2, ap_confidence_interval_l2 =
plot_roc_pr_curves(y_val, y_val_probs_l2)

# Predict on the test set
y_test_pred_l2 = best_log_reg_l2.predict(X_test)

# Predict probabilities on the test set
y_test_probs_l2 = best_log_reg_l2.predict_proba(X_test)[:, 1]

# AUROC and PR curves with bootstrapped 95% confidence intervals for
# logistic regression with L2 regularization after hyperparameter tuning on test set
auroc_confidence_interval_l2, ap_confidence_interval_l2 =
plot_roc_pr_curves(y_test, y_test_probs_l2)

# Generating the confusion matrix
cm = confusion_matrix(y_test, y_test_pred_l2)
test_accuracy = accuracy_score(y_test, y_test_pred_l2)
print(f"Test Accuracy: {test_accuracy:.4f}")
print(cm)
print(classification_report(y_test, y_test_pred_l2))
```

```
# Plotting the confusion matrix
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues',
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()

# Feature importance
# Get the coefficients from the L1 model
log_reg_l1_coefficients = best_log_reg_l1.coef_.flatten()

# Get the coefficients from the L2 model
log_reg_l2_coefficients = best_log_reg_l2.coef_.flatten()

# For better interpretability, look at the absolute values of the coefficients
log_reg_l1_importance = np.abs(log_reg_l1_coefficients)
log_reg_l2_importance = np.abs(log_reg_l2_coefficients)

# Create a DataFrame for easy viewing of feature importance for both models
import pandas as pd

feature_names = X_train.columns
importance_df = pd.DataFrame({
    'Feature': feature_names,
    'L1 Coefficient': log_reg_l1_coefficients,
    'L1 Importance': log_reg_l1_importance,
    'L2 Coefficient': log_reg_l2_coefficients,
    'L2 Importance': log_reg_l2_importance,
}).sort_values('L1 Importance', ascending=False)

print(importance_df)
```

2.2 Breast Cancer SVM Code

```
import sys
sys.path.append('../')
from imports import *
from sklearn.svm import SVC

def plot_roc_pr_curves(y_true, y_probs, n_bootstraps=1000):
    """Perform bootstrapping to calculate 95% confidence intervals for
    ROC and PR curves and plot them"""
    bootstrap_auroc_scores = []
    bootstrap_average_precision_scores = []

    for _ in range(n_bootstraps):
        # Bootstrap sample (with replacement)
        indices = resample(np.arange(len(y_true)), replace=True)
```

```
            y_true_boot = y_true[indices]
            y_probs_boot = y_probs[indices]

            # Compute metrics for bootstrap sample
            bootstrap_auroc_scores.append(roc_auc_score(y_true_boot, y_probs_boot))
            bootstrap_average_precision_scores.append(average_precision_score(
            y_true_boot, y_probs_boot))

        # Compute confidence intervals
        auroc_lower = np.percentile(bootstrap_auroc_scores, 2.5)
        auroc_upper = np.percentile(bootstrap_auroc_scores, 97.5)
        ap_lower = np.percentile(bootstrap_average_precision_scores, 2.5)
        ap_upper = np.percentile(bootstrap_average_precision_scores, 97.5)

        # Calculate original ROC and PR curves
        fpr, tpr, _ = roc_curve(y_true, y_probs)
        precision, recall, _ = precision_recall_curve(y_true, y_probs)
        auroc = roc_auc_score(y_true, y_probs)
        average_precision = average_precision_score(y_true, y_probs)

        # Plotting
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

        # ROC Curve
        ax1.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {auroc:.2f})')
        ax1.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
        ax1.set_xlabel('False Positive Rate')
        ax1.set_ylabel('True Positive Rate')
        ax1.set_title('Receiver Operating Characteristic')
        ax1.legend(loc="lower right", title=f'95% CI: [{auroc_lower:.2f}, {auroc_upper:.2f}]')

        # Precision-Recall Curve
        ax2.plot(recall, precision, color='blue', lw=2, label=f'PR curve
        (area = {average_precision:.2f})')
        ax2.set_xlabel('Recall')
        ax2.set_ylabel('Precision')
        ax2.set_title('Precision-Recall Curve')
        ax2.legend(loc="lower left", title=f'95% CI: [{ap_lower:.2f}, {ap_upper:.2f}]')

        plt.show()

        # Return the confidence intervals
        return (np.round(auroc_lower,2), np.round(auroc_upper,2)),
        (np.round(ap_lower,2), np.round(ap_upper,2))

# Loading the datasets
X_train = pd.read_csv('./breast-cancer-processed-data/X_train.csv')
y_train = pd.read_csv('./breast-cancer-processed-data/y_train.csv')
X_val = pd.read_csv('./breast-cancer-processed-data/X_val.csv')
y_val = pd.read_csv('./breast-cancer-processed-data/y_val.csv')
X_test = pd.read_csv('./breast-cancer-processed-data/X_test.csv')
y_test = pd.read_csv('./breast-cancer-processed-data/y_test.csv')

y_train = y_train.squeeze()
```

```
y_val = y_val.squeeze()
y_test = y_test.squeeze()

# Initialize the SVM model without kernel
svm = SVC(kernel='linear', probability=True, random_state=1)

# Train the model on the training set
svm.fit(X_train, y_train)

# Predict on the validation set
y_val_pred = svm.predict(X_val)
print(confusion_matrix(y_val, y_val_pred))
print(classification_report(y_val, y_val_pred))

# Evaluate the model on the validation set
val_accuracy = accuracy_score(y_val, y_val_pred)
print(f"Validation Set Accuracy: {val_accuracy:.4f}")

# Predict probabilities
y_val_probs = svm.predict_proba(X_val)[:, 1]

# Compute AUROC
auroc = roc_auc_score(y_val, y_val_probs)
print(f"Area under the ROC curve: {auroc:.2f}")

# Compute Precision-Recall curve and its area
average_precision = average_precision_score(y_val, y_val_probs)
print(f"Area under the Precision-Recall curve: {average_precision:.2f}")

# Initialize the SVM model with polynomial kernel
svm_poly = SVC(kernel='poly', probability=True, random_state=1)

# Train the model on the training set
svm_poly.fit(X_train, y_train)

# Predict on the validation set
y_val_pred_poly = svm_poly.predict(X_val)
print(confusion_matrix(y_val, y_val_pred_poly))
print(classification_report(y_val, y_val_pred_poly))

# Evaluate the model on the validation set
val_accuracy = accuracy_score(y_val, y_val_pred_poly)
print(f"Validation Set Accuracy: {val_accuracy:.4f}")

# Predict probabilities
y_val_probs_poly = svm_poly.predict_proba(X_val)[:, 1]

# Compute AUROC
auroc = roc_auc_score(y_val, y_val_probs_poly)
print(f"Area under the ROC curve: {auroc:.2f}")

# Compute Precision-Recall curve and its area
average_precision = average_precision_score(y_val, y_val_probs_poly)
print(f"Area under the Precision-Recall curve: {average_precision:.2f}")
```

```
# Initialize the SVM model without kernel
svm_rbf = SVC(kernel='rbf', probability=True, random_state=1)

# Train the model on the training set
svm_rbf.fit(X_train, y_train)

# Predict on the validation set
y_val_pred_rbf = svm_rbf.predict(X_val)
print(confusion_matrix(y_val, y_val_pred_rbf))
print(classification_report(y_val, y_val_pred_rbf))

# Evaluate the model on the validation set
val_accuracy = accuracy_score(y_val, y_val_pred_rbf)
print(f"Validation Set Accuracy: {val_accuracy:.4f}")

# Predict probabilities
y_val_probs_rbf = svm_rbf.predict_proba(X_val)[:, 1]

# Compute AUROC
auroc = roc_auc_score(y_val, y_val_probs_rbf)
print(f"Area under the ROC curve: {auroc:.2f}")

# Compute Precision-Recall curve and its area
average_precision = average_precision_score(y_val, y_val_probs_rbf)
print(f"Area under the Precision-Recall curve: {average_precision:.2f}")

# Predict on the test set
y_test_pred = svm_rbf.predict(X_test)

# Predict probabilities on the test set
y_test_probs = svm_rbf.predict_proba(X_test)[:, 1]

# Compute AUROC
auroc = roc_auc_score(y_test, y_test_probs)
print(f"Area under the ROC curve: {auroc:.2f}")

# Compute Precision-Recall curve and its area
average_precision = average_precision_score(y_test, y_test_probs)
print(f"Area under the Precision-Recall curve: {average_precision:.2f}")

# Generating the confusion matrix
cm = confusion_matrix(y_test, y_test_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test Accuracy: {test_accuracy:.4f}")
print(cm)
print(classification_report(y_test, y_test_pred))


# Plotting the confusion matrix
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues',
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])
```

```
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()

# Plot AUROC and PR curves on test set with 95% confidence intervals
plot_roc_pr_curves(y_test, y_test_probs)
```

2.3 Breast Cancer PCA+KNN Code

```
import sys
sys.path.append('../')
from imports import *

# Loading the datasets
X_train = pd.read_csv('./breast-cancer-processed-data/X_train.csv')
y_train = pd.read_csv('./breast-cancer-processed-data/y_train.csv')
X_val = pd.read_csv('./breast-cancer-processed-data/X_val.csv')
y_val = pd.read_csv('./breast-cancer-processed-data/y_val.csv')
X_test = pd.read_csv('./breast-cancer-processed-data/X_test.csv')
y_test = pd.read_csv('./breast-cancer-processed-data/y_test.csv')

y_train = y_train.squeeze()
y_val = y_val.squeeze()
y_test = y_test.squeeze()

# Step 1: Apply PCA
# Choose the number of components, e.g., 2 for visualization
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)
X_val_pca = pca.transform(X_val)
X_test_pca = pca.transform(X_test)

# Step 2: Train KNN Classifier
# Choose the number of neighbors, e.g., 3
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train_pca, y_train)

# Step 3: Predict and Evaluate the model
y_val_pred = knn.predict(X_val_pca)
val_accuracy = accuracy_score(y_val, y_val_pred)

print(f"Validation Accuracy: {val_accuracy:.4f}")

print(confusion_matrix(y_val, y_val_pred))
print(classification_report(y_val, y_val_pred))

# Create a pipeline that first applies PCA and then KNN
pipe = Pipeline([
    ('pca', PCA()),
    ('knn', KNeighborsClassifier())
])
```

```
# Define the parameter grid to search
param_grid = {
    'pca__n_components': [2, 5, 10, 15, 20, 25, 30],  # Example values
    'knn__n_neighbors': [1, 3, 5, 7, 9, 11, 13]      # Example values
}

# Create a GridSearchCV object
grid_search = GridSearchCV(pipe, param_grid, cv=5, verbose=2)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Print the best parameters and the corresponding score
print("\nBest parameters:", grid_search.best_params_)
print("Best cross-validation score:", grid_search.best_score_)

# now same pipeline with the best parameters
pca = PCA(n_components=20)
X_train_pca = pca.fit_transform(X_train)
X_val_pca = pca.transform(X_val)
X_test_pca = pca.transform(X_test)

# Step 2: Train KNN Classifier
knn = KNeighborsClassifier(n_neighbors=9)
knn.fit(X_train_pca, y_train)

# Step 3: Predict and Evaluate the model
y_val_pred = knn.predict(X_val_pca)
val_accuracy = accuracy_score(y_val, y_val_pred)

print(f"Validation Accuracy: {val_accuracy:.4f}")

print(confusion_matrix(y_val, y_val_pred))
print(classification_report(y_val, y_val_pred))

# Evaluate the model on test data
# Use the KNN model to predict on the test set
y_test_pred = knn.predict(X_test_pca)

# Generating the confusion matrix
cm = confusion_matrix(y_test, y_test_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test Accuracy: {test_accuracy:.4f}")
print(cm)
print(classification_report(y_test, y_test_pred))


# Plotting the confusion matrix
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues',
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
```

```
plt.title('Confusion Matrix')
plt.show()
```

2.4 Breast Cancer FNN Code

```
#%%
import sys

import numpy as np

sys.path.append('../')
from imports import *
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import ParameterGrid
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve, auc
import seaborn as sns


#%%
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
#%%
torch.cuda.is_available()
#%%
X_train = pd.read_csv('./breast-cancer-processed-data/X_train.csv').values
y_train = pd.read_csv('./breast-cancer-processed-data/y_train.csv').values
X_val = pd.read_csv('./breast-cancer-processed-data/X_val.csv').values
y_val = pd.read_csv('./breast-cancer-processed-data/y_val.csv').values
X_test = pd.read_csv('./breast-cancer-processed-data/X_test.csv').values
y_test = pd.read_csv('./breast-cancer-processed-data/y_test.csv').values
#%%

#%%
# Convert Numpy arrays to PyTorch tensors
X_train, y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dtype=
X_val, y_val = torch.tensor(X_val, dtype=torch.float32), torch.tensor(y_val, dtype=torch.fl
X_test, y_test = torch.tensor(X_test, dtype=torch.float32), torch.tensor(y_test, dtype=torc

# Create dataloaders
train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=True)
val_loader = DataLoader(TensorDataset(X_val, y_val), batch_size=32)
test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=1)
#%%

#%%

#%%
```

```
#%%
class FNN(nn.Module):
    def __init__(self, input_size):
        super(FNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(64, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.sigmoid(x)
        return x
#%%
model = FNN(X_train.shape[1]).to(device)
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.0005)

#%%
def calculate_accuracy(y_pred, y_true):
    # Convert predictions to binary values (0 or 1)

    predicted = torch.round(y_pred).view(-1)
    y_true = y_true.squeeze()
    return (predicted == y_true).sum().float() / len(y_true)

train_losses, val_losses = [], []
train_accuracies, val_accuracies = [], []
# Training and validation for certain epochs
epochs = 100
for epoch in range(epochs):  # number of epochs
    model.train()
    total_loss, total_acc = 0, 0
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        total_acc += calculate_accuracy(output, target)

    avg_train_loss = total_loss / len(train_loader)
    avg_train_acc = total_acc / len(train_loader)
    train_losses.append(avg_train_loss)
    train_accuracies.append(avg_train_acc)

    # Validation
    model.eval()
    val_loss, val_acc = 0, 0
    with torch.no_grad():
```

```
        for data, target in val_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            val_loss += criterion(output, target).item()
            val_acc += calculate_accuracy(output, target)

    avg_val_loss = val_loss / len(val_loader)
    avg_val_acc = val_acc / len(val_loader)
    val_losses.append(avg_val_loss)
    val_accuracies.append(avg_val_acc)
    print(f'Epoch {epoch+1}, Training Loss: {avg_train_loss:.4f}, Training Accuracy: {avg_


#%%

#%%
import matplotlib.pyplot as plt

# Plotting training and validation losses
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.title('Training & Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
train_accuracies = [acc.cpu().numpy() for acc in train_accuracies]
val_accuracies = [acc.cpu().numpy() for acc in val_accuracies]

# Plotting training and validation accuracies
plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Training Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.title('Training & Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()

#%%
# Testing the model
model.eval()
test_loss = 0
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        output = model(data)
        test_loss += criterion(output, target).item()

        # For accuracy calculation
```

```
        predicted = torch.round(output).view(-1)  # Assuming binary classification
        correct += (predicted == target).sum().item()
        total += target.size(0)

test_loss /= len(test_loader)
test_accuracy = 100 * correct / total
print(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2f}%')

#%%
# Now let us perform hyperparamter tuning to find optimal parameters and replace them where
# Defining the parametr grid for hyperparameter tuning
param_grid = {
    'lr': [0.0001, 0.0005, 0.005, 0.001, 0.01],
    'batch_size': [16, 32, 64],
    'epochs': [50, 100]
}
#%%
def calculate_accuracy(y_pred, y_true):
    # Convert predictions to binary values (0 or 1)
    y_pred, y_true = y_pred.squeeze(), y_true.squeeze()
    predicted = torch.round(y_pred).view(-1)
    return (predicted == y_true).sum().float() / len(y_true)
#%%
def train_and_validate(model, train_loader, val_loader, criterion, optimizer, epochs):
    train_losses, val_losses = [], []
    train_accuracies, val_accuracies = [], []

    for epoch in range(epochs):
        model.train()
        total_loss, total_acc = 0, 0
        for data, target in train_loader:
            data, target = data.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
            total_acc += calculate_accuracy(output, target)

        avg_train_loss = total_loss / len(train_loader)
        avg_train_acc = total_acc / len(train_loader)
        train_losses.append(avg_train_loss)
        train_accuracies.append(avg_train_acc)

        model.eval()
        val_loss, val_acc = 0, 0
        with torch.no_grad():
            for data, target in val_loader:
                data, target = data.to(device), target.to(device)
                output = model(data)
                val_loss += criterion(output, target).item()
                val_acc += calculate_accuracy(output, target)
```

```
        avg_val_loss = val_loss / len(val_loader)
        avg_val_acc = val_acc / len(val_loader)
        val_losses.append(avg_val_loss)
        val_accuracies.append(avg_val_acc)

    return avg_val_loss, avg_val_acc

#%%
best_val_acc = 0
best_params = {}

for params in ParameterGrid(param_grid):
    train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=params['batch_siz
    val_loader = DataLoader(TensorDataset(X_val, y_val), batch_size=params['batch_size'])

    model = FNN(X_train.shape[1]).to(device)
    criterion = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=params['lr'])

    val_loss, val_acc = train_and_validate(model, train_loader, val_loader, criterion, opti

    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_params = params

    print(f"Params: {params}, Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_ac

print(f"Best Params: {best_params}, Best Validation Accuracy: {best_val_acc:.4f}%")

#%%

#%%
model = FNN(X_train.shape[1]).to(device)
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.0005)
# Convert Numpy arrays to PyTorch tensors
X_train, y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dtype=
X_val, y_val = torch.tensor(X_val, dtype=torch.float32), torch.tensor(y_val, dtype=torch.fl
X_test, y_test = torch.tensor(X_test, dtype=torch.float32), torch.tensor(y_test, dtype=torc

# Create dataloaders
train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=64, shuffle=True)
val_loader = DataLoader(TensorDataset(X_val, y_val), batch_size=64)
test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=1)

def calculate_accuracy(y_pred, y_true):
    # Convert predictions to binary values (0 or 1)

    predicted = torch.round(y_pred).view(-1)
    y_true = y_true.squeeze()
    return (predicted == y_true).sum().float() / len(y_true)


train_losses, val_losses = [], []
```

```
train_accuracies, val_accuracies = [], []
# Training and validation for certain epochs
epochs = 50
for epoch in range(epochs):  # number of epochs
    model.train()
    total_loss, total_acc = 0, 0
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        total_acc += calculate_accuracy(output, target)

    avg_train_loss = total_loss / len(train_loader)
    avg_train_acc = total_acc / len(train_loader)
    train_losses.append(avg_train_loss)
    train_accuracies.append(avg_train_acc)

    # Validation
    model.eval()
    val_loss, val_acc = 0, 0
    with torch.no_grad():
        for data, target in val_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            val_loss += criterion(output, target).item()
            val_acc += calculate_accuracy(output, target)

    avg_val_loss = val_loss / len(val_loader)
    avg_val_acc = val_acc / len(val_loader)
    val_losses.append(avg_val_loss)
    val_accuracies.append(avg_val_acc)
    print(
        f'Epoch {epoch + 1}, Training Loss: {avg_train_loss:.4f}, Training Accuracy: {avg_


#%%

# Plotting training and validation losses
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.title('Training & Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
train_accuracies = [acc.cpu().numpy() for acc in train_accuracies]
val_accuracies = [acc.cpu().numpy() for acc in val_accuracies]

# Plotting training and validation accuracies
```

```
plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Training Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.title('Training & Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()



#%%
# Testing the model
model.eval()
test_loss = 0
correct = 0
total = 0
y_true = y_test
y_pred = []
y_scores = []

with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device).squeeze(1)
        output = model(data).squeeze(1)
        y_scores.extend(output.cpu().numpy().flatten())
        test_loss += criterion(output, target).item()

        # For accuracy calculation
        predicted = torch.round(output).view(-1)  # Assuming binary classification
        y_pred.append(predicted)

        correct += (predicted == target).sum().item()
        total += target.size(0)
fpr, tpr, _ = roc_curve(y_true, y_scores)
roc_auc = auc(fpr, tpr)
test_loss /= len(test_loader)
test_accuracy = 100 * correct / total
print(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2f}%')
#%%
y_true = np.array(y_true)
#%%
y_pred = [pred.to('cpu') for pred in y_pred]
#%%
y_pred = np.array(y_pred)
#%%
y_pred.shape
#%%

# Calculate the confusion matrix
cm = confusion_matrix(y_true, y_pred)
print(cm)
# Plot the confusion matrix
```

```
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='g', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

#%%
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic for Neural Network')
plt.legend(loc="lower right")
plt.show()
#%%
```

3.1 UCI Adult Logistic Regression Code

```
import sys
sys.path.append('../')
from imports import *
import cudf
import cupy as cp
from sklearn.utils import resample

def plot_roc_pr_curves(y_true, y_probs, n_bootstraps=1000):
    """Perform bootstrapping to calculate 95% confidence intervals for
    ROC and PR curves and plot them"""
    bootstrap_auroc_scores = []
    bootstrap_average_precision_scores = []

    for _ in range(n_bootstraps):
        # Bootstrap sample (with replacement)
        indices = resample(np.arange(len(y_true)), replace=True)
        y_true_boot = y_true[indices]
        y_probs_boot = y_probs[indices]

        # Compute metrics for bootstrap sample
        bootstrap_auroc_scores.append(roc_auc_score(y_true_boot, y_probs_boot))
        bootstrap_average_precision_scores.append(average_precision_score(
        y_true_boot, y_probs_boot))

    # Compute confidence intervals
    auroc_lower = np.percentile(bootstrap_auroc_scores, 2.5)
    auroc_upper = np.percentile(bootstrap_auroc_scores, 97.5)
    ap_lower = np.percentile(bootstrap_average_precision_scores, 2.5)
    ap_upper = np.percentile(bootstrap_average_precision_scores, 97.5)

    # Calculate original ROC and PR curves
```

```
    fpr, tpr, _ = roc_curve(y_true, y_probs)
    precision, recall, _ = precision_recall_curve(y_true, y_probs)
    auroc = roc_auc_score(y_true, y_probs)
    average_precision = average_precision_score(y_true, y_probs)

    # Plotting
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

    # ROC Curve
    ax1.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {auroc:.2f})')
    ax1.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    ax1.set_xlabel('False Positive Rate')
    ax1.set_ylabel('True Positive Rate')
    ax1.set_title('Receiver Operating Characteristic')
    ax1.legend(loc="lower right", title=f'95% CI: [{auroc_lower:.2f}, {auroc_upper:.2f}]')

    # Precision-Recall Curve
    ax2.plot(recall, precision, color='blue', lw=2, label=f'PR curve
    (area = {average_precision:.2f})')
    ax2.set_xlabel('Recall')
    ax2.set_ylabel('Precision')
    ax2.set_title('Precision-Recall Curve')
    ax2.legend(loc="lower left", title=f'95% CI: [{ap_lower:.2f}, {ap_upper:.2f}]')

    plt.show()

    # Return the confidence intervals
    return (np.round(auroc_lower,2), np.round(auroc_upper,2)),
    (np.round(ap_lower,2), np.round(ap_upper,2))

# loading processed data
df_train = pd.read_csv('./uci-adult-processed-data/train.csv')
df_val = pd.read_csv('./uci-adult-processed-data/val.csv')
df_test = pd.read_csv('./uci-adult-processed-data/test.csv')

# Define your features and target variable
target_column = 'target'
features = df_train.columns.drop(target_column)

# Separate features and target and convert to NumPy arrays
X_train = df_train[features].to_numpy()
y_train = df_train[target_column].to_numpy()
X_val = df_val[features].to_numpy()
y_val = df_val[target_column].to_numpy()
X_test = df_test[features].to_numpy()
y_test = df_test[target_column].to_numpy()

# Normalize the data
# Initialize the StandardScaler
scaler = StandardScaler()
# Fit the scaler on the training data and transform the training data
X_train = scaler.fit_transform(X_train)

# Transform the validation and test data
```

```
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)

# Initialize and fit the logistic regression model
log_reg = LogisticRegression(max_iter=1000, class_weight='balanced')
log_reg.fit(X_train, y_train)

# Predict on validation set
y_val_pred = log_reg.predict(X_val)

# Evaluate the model
val_accuracy = accuracy_score(y_val, y_val_pred)
print("Validation Accuracy:", val_accuracy)
print("Validation Classification Report:\n", classification_report(y_val, y_val_pred))

# Predict probabilities
y_val_probs = log_reg.predict_proba(X_val)[:, 1]

# Compute AUROC
auroc = roc_auc_score(y_val, y_val_probs)
print(f"Area under the ROC curve: {auroc:.2f}")

# Compute Precision-Recall curve and its area
average_precision = average_precision_score(y_val, y_val_probs)
print(f"Area under the Precision-Recall curve: {average_precision:.2f}")

# AUROC and PR curves with bootstrapped 95% confidence intervals
# for logistic regression with no regularization
auroc_confidence_interval, ap_confidence_interval =
plot_roc_pr_curves(y_val, y_val_probs)

# Initialize the Logistic Regression model with L1 regularization
# You may need to tune the 'C' parameter using the validation set
log_reg_l1 = LogisticRegression(penalty='l1', C=1.0, solver='liblinear', max_iter=1000, cla

# Train the model on the training set
log_reg_l1.fit(X_train, y_train)

# Predict probabilities for the validation and test sets
y_val_probs_l1 = log_reg_l1.predict_proba(X_val)[:, 1]
y_test_probs_l1 = log_reg_l1.predict_proba(X_test)[:, 1]

# Compute the AUROC and average precision score on the validation set
val_auroc = roc_auc_score(y_val, y_val_probs_l1)
val_average_precision = average_precision_score(y_val, y_val_probs_l1)

# Compute the AUROC and average precision score on the test set
test_auroc = roc_auc_score(y_test, y_test_probs_l1)
test_average_precision = average_precision_score(y_test, y_test_probs_l1)

# AUROC and PR curves with bootstrapped 95% confidence intervals
# for logistic regression with no regularization
auroc_confidence_interval, ap_confidence_interval =
plot_roc_pr_curves(y_val, y_val_probs_l1)
```

```
# Define a set of C values to try
C_values = [0.001, 0.01, 0.1, 1, 10, 100]

# Initialize variables to store the best score and corresponding C value
best_score = 0
best_C = None

# Perform grid search over the C values
for C in C_values:
    # Initialize and train the Logistic Regression model with L1 regularization
    log_reg_l1 = LogisticRegression(penalty='l1', C=C,
    solver='liblinear', class_weight='balanced', max_iter=1000, random_state=1)
    log_reg_l1.fit(X_train, y_train)

    # Evaluate on the validation set
    y_val_probs = log_reg_l1.predict_proba(X_val)[:, 1]
    score = roc_auc_score(y_val, y_val_probs)

    # If the score is better than the best score, update the best score and best C
    if score > best_score:
        best_score = score
        best_C = C

# Output the best C value
print(f"Best C value: {best_C} with AUROC: {best_score:.4f}")

# Train a new model using the best C value
best_log_reg_l1 = LogisticRegression(penalty='l1', C=best_C,
solver='liblinear', max_iter=1000, random_state=1)
best_log_reg_l1.fit(X_train, y_train)

# Continue with bootstrapping, plotting, and evaluation using the best_log_reg_l1 model

# Predict probabilities for the validation and test sets
y_val_probs_l1 = best_log_reg_l1.predict_proba(X_val)[:, 1]

# AUROC and PR curves with bootstrapped 95% confidence intervals
# for logistic regression with L1 regularization
auroc_confidence_interval_l1, ap_confidence_interval_l1 =
plot_roc_pr_curves(y_val, y_val_probs_l1)

# Define a logistic regression model. Note: The default penalty is 'l2'.
log_reg_l2 = LogisticRegression(solver='liblinear',
class_weight='balanced', random_state=1)

# Define the hyperparameter grid for 'C'
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

# Define the scoring function you want to optimize for during the grid search
scorer = make_scorer(roc_auc_score, needs_proba=True)

# Set up the grid search with cross-validation
grid_search = GridSearchCV(log_reg_l2, param_grid, cv=5, scoring=scorer)
```

```
# Perform grid search on the training set
grid_search.fit(X_train, y_train)

# Output the best C value
print(f"Best parameters: {grid_search.best_params_}")

# After grid search, the best hyperparameter value is stored in 'best_estimator_'
best_log_reg_l2 = grid_search.best_estimator_

# Now using 'best_l2_model' to make predictions and evaluate it
y_val_probs_l2 = best_log_reg_l2.predict_proba(X_val)[:, 1]

# Evaluate the model using the validation  sets
val_auroc_l2 = roc_auc_score(y_val, y_val_probs_l2)

# AUROC and PR curves with bootstrapped 95% confidence intervals
# for logistic regression with L2 regularization after hyperparameter tuning
auroc_confidence_interval_l2, ap_confidence_interval_l2 =
plot_roc_pr_curves(y_val, y_val_probs_l2)

# Predict on the validation set
y_val_pred_l1 = best_log_reg_l1.predict(X_val)

# Predict probabilities on the validation set
y_val_probs_l1 = best_log_reg_l1.predict_proba(X_val)[:, 1]

# Predict on the test set
y_test_pred_l1 = best_log_reg_l1.predict(X_test)

# Predict probabilities on the test set
y_test_probs_l1 = best_log_reg_l1.predict_proba(X_test)[:, 1]

# AUROC and PR curves with bootstrapped 95% confidence intervals for logistic regression
#  with L1 regularization after hyperparameter tuning on val set
auroc_confidence_interval_l1, ap_confidence_interval_l1 =
plot_roc_pr_curves(y_val, y_val_probs_l1)

# AUROC and PR curves with bootstrapped 95% confidence intervals for logistic regression
# with L1 regularization after hyperparameter tuning on test set
auroc_confidence_interval_l1, ap_confidence_interval_l1 =
plot_roc_pr_curves(y_test, y_test_probs_l1)

# Generating the confusion matrix
cm = confusion_matrix(y_val, y_val_pred_l1)
val_accuracy = accuracy_score(y_val, y_val_pred_l1)
print(f"Val Accuracy: {val_accuracy:.4f}")
print(cm)
print(classification_report(y_val, y_val_pred_l1))

# Generating the confusion matrix
cm = confusion_matrix(y_test, y_test_pred_l1)
test_accuracy = accuracy_score(y_test, y_test_pred_l1)
print(f"Test Accuracy: {test_accuracy:.4f}")
```

```
print(cm)
print(classification_report(y_test, y_test_pred_l1))


# Plotting the confusion matrix
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues',
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()

# Get the coefficients from the L1 model
log_reg_l1_coefficients = best_log_reg_l1.coef_.flatten()

# For better interpretability, look at the absolute values of the coefficients
log_reg_l1_importance = np.abs(log_reg_l1_coefficients)

importance_df = pd.DataFrame({
    'Feature': features,
    'L1 Coefficient': log_reg_l1_coefficients,
    'L1 Importance': log_reg_l1_importance,
}).sort_values('L1 Importance', ascending=False)

print(importance_df)
```

3.2 UCI Adult SVM Code

```
import sys
sys.path.append('../')
from imports import *
# from sklearn.svm import SVC
from sklearn.utils import resample
import cudf
import cupy as cp
import cuml
from cuml.svm import SVC, LinearSVC

def plot_roc_pr_curves(y_true, y_probs, n_bootstraps=1000):
    """Perform bootstrapping to calculate 95% confidence intervals
    for ROC and PR curves and plot them"""
    bootstrap_auroc_scores = []
    bootstrap_average_precision_scores = []

    for _ in range(n_bootstraps):
        # Bootstrap sample (with replacement)
        indices = resample(np.arange(len(y_true)), replace=True)
        y_true_boot = y_true[indices]
        y_probs_boot = y_probs[indices]

        # Compute metrics for bootstrap sample
```

```
            bootstrap_auroc_scores.append(roc_auc_score
            (y_true_boot, y_probs_boot))
            bootstrap_average_precision_scores.append
            (average_precision_score(y_true_boot, y_probs_boot))

        # Compute confidence intervals
        auroc_lower = np.percentile(bootstrap_auroc_scores, 2.5)
        auroc_upper = np.percentile(bootstrap_auroc_scores, 97.5)
        ap_lower = np.percentile(bootstrap_average_precision_scores, 2.5)
        ap_upper = np.percentile(bootstrap_average_precision_scores, 97.5)

        # Calculate original ROC and PR curves
        fpr, tpr, _ = roc_curve(y_true, y_probs)
        precision, recall, _ = precision_recall_curve(y_true, y_probs)
        auroc = roc_auc_score(y_true, y_probs)
        average_precision = average_precision_score(y_true, y_probs)

        # Plotting
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

        # ROC Curve
        ax1.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {auroc:.2f})')
        ax1.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
        ax1.set_xlabel('False Positive Rate')
        ax1.set_ylabel('True Positive Rate')
        ax1.set_title('Receiver Operating Characteristic')
        ax1.legend(loc="lower right", title=f'95% CI: [{auroc_lower:.2f}, {auroc_upper:.2f}]')

        # Precision-Recall Curve
        ax2.plot(recall, precision, color='blue', lw=2, label=f'PR curve
        (area = {average_precision:.2f})')
        ax2.set_xlabel('Recall')
        ax2.set_ylabel('Precision')
        ax2.set_title('Precision-Recall Curve')
        ax2.legend(loc="lower left", title=f'95% CI: [{ap_lower:.2f}, {ap_upper:.2f}]')

        plt.show()

        # Return the confidence intervals
        return (np.round(auroc_lower,2), np.round(auroc_upper,2)),
        (np.round(ap_lower,2), np.round(ap_upper,2))

# loading processed data
df_train = pd.read_csv('./uci-adult-processed-data/train.csv')
df_val = pd.read_csv('./uci-adult-processed-data/val.csv')
df_test = pd.read_csv('./uci-adult-processed-data/test.csv')

# Define your features and target variable
target_column = 'target'
features = df_train.columns.drop(target_column)

# Separate features and target and convert to NumPy arrays
X_train = df_train[features].to_numpy()
y_train = df_train[target_column].to_numpy()
```

```
X_val = df_val[features].to_numpy()
y_val = df_val[target_column].to_numpy()
X_test = df_test[features].to_numpy()
y_test = df_test[target_column].to_numpy()

# Initialize the StandardScaler
scaler = StandardScaler()
# Fit the scaler on the training data and transform the training data
X_train = scaler.fit_transform(X_train)

# Transform the validation and test data
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)

# convert to cupy arrays
X_train_cp = cp.array(X_train, dtype=cp.float32)
y_train_cp = cp.array(y_train, dtype=cp.float32)
X_val_cp = cp.array(X_val, dtype=cp.float32)
y_val_cp = cp.array(y_val, dtype=cp.float32)
X_test_cp = cp.array(X_test, dtype=cp.float32)
y_test_cp = cp.array(y_test, dtype=cp.float32)

# calculate the class weights
num_zeros = np.sum(y_train == 0)
num_ones = np.sum(y_train == 1)

total = y_train.size
percent_zeros = (num_zeros / total) * 100
percent_ones = (num_ones / total) * 100

print(f"Percentage of 0's: {percent_zeros:.2f}%")
print(f"Percentage of 1's: {percent_ones:.2f}%")

# Set class weights
class_weights = {0: 1, 1: 3}  # Since class 1 is
# approximately 3 times less frequent than class 0

# Initialize the SVM model without kernel
svm_linear = LinearSVC(loss='squared_hinge', penalty='l1',
C=1, probability=True, class_weight='balanced', output_type='numpy')

# Train the model on the training set
svm_linear.fit(X_train_cp, y_train_cp)

# Predict and evaluate
y_val_pred = svm_linear.predict(X_val)

# Extract the probabilities for the positive class
y_val_probs = svm_linear.predict_proba(X_val)[:, 1]

# confusion matrix and classification report
print(confusion_matrix(y_val, y_val_pred))
print(classification_report(y_val, y_val_pred))
```

```
# Evaluate the model on the validation set
val_accuracy = accuracy_score(y_val, y_val_pred)
print(f"Validation Set Accuracy: {val_accuracy:.4f}")

# Compute AUROC
auroc = roc_auc_score(y_val, y_val_probs)
print(f"Area under the ROC curve: {auroc:.4f}")

# Compute Precision-Recall curve and its area
average_precision = average_precision_score(y_val, y_val_probs)
print(f"Area under the Precision-Recall curve: {average_precision:.4f}")

# Plot ROC and PR curves for linear SVC
plot_roc_pr_curves(y_val, y_val_probs)

# Initialize the SVM model with polynomial kernel
svm_poly = SVC(kernel='poly', probability=True,
class_weight='balanced', output_type='numpy')

# Train the model on the training set
svm_poly.fit(X_train_cp, y_train_cp)

# Predict and evaluate on validation set
y_val_poly_pred = svm_poly.predict(X_val)

# Extract the probabilities for the positive class
y_val_poly_prob = svm_poly.predict_proba(X_val)[:, 1]

print(confusion_matrix(y_val, y_val_poly_pred))
print(classification_report(y_val, y_val_poly_pred))

# Evaluate the model on the validation set
val_accuracy = accuracy_score(y_val, y_val_poly_pred)
print(f"Validation Accuracy: {val_accuracy:.4f}")

# Plot ROC and PR curves for polynomial SVC with 95% confidence intervals
plot_roc_pr_curves(y_val, y_val_poly_prob)

# Initialize the SVM model with rbf kernel
svm_rbf = SVC(kernel='rbf', probability=True, class_weight='balanced', output_type='numpy')

# Train the model on the training set
svm_rbf.fit(X_train_cp, y_train_cp)

# Predict and evaluate
y_val_rbf_pred = svm_rbf.predict(X_val)

# Extract the probabilities for the positive class
y_val_rbf_prob = svm_rbf.predict_proba(X_val)[:, 1]

print(confusion_matrix(y_val, y_val_rbf_pred))
print(classification_report(y_val, y_val_rbf_pred))

# Evaluate the model on the validation set
```

```
val_accuracy = accuracy_score(y_val, y_val_rbf_pred)
print(f"Validation Accuracy: {val_accuracy:.4f}")

# Plot ROC and PR curves for rbf SVC with 95% confidence intervals
plot_roc_pr_curves(y_val, y_val_rbf_prob)

# Predict on the test set
y_test_pred = svm_rbf.predict(X_test)

# Predict probabilities on the test set
y_test_probs = svm_rbf.predict_proba(X_test)[:, 1]

# Compute AUROC
auroc = roc_auc_score(y_test, y_test_probs)
print(f"Area under the ROC curve: {auroc:.2f}")

# Compute Precision-Recall curve and its area
average_precision = average_precision_score(y_test, y_test_probs)
print(f"Area under the Precision-Recall curve: {average_precision:.2f}")

# Generating the confusion matrix
cm = confusion_matrix(y_test, y_test_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test Accuracy: {test_accuracy:.4f}")
print(cm)
print(classification_report(y_test, y_test_pred))


# Plotting the confusion matrix
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues',
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()

# Plot AUROC and PR curves on test set with 95% confidence intervals
plot_roc_pr_curves(y_test, y_test_probs)

# Predict on the test set
y_test_pred_1 = svm_linear.predict(X_test)

# Predict probabilities on the test set
y_test_probs_1 = svm_linear.predict_proba(X_test)[:, 1]

# Compute AUROC
auroc = roc_auc_score(y_test, y_test_probs_1)
print(f"Area under the ROC curve: {auroc:.2f}")

# Compute Precision-Recall curve and its area
average_precision = average_precision_score(y_test, y_test_probs_1)
print(f"Area under the Precision-Recall curve: {average_precision:.2f}")
```

```
# Generating the confusion matrix
cm = confusion_matrix(y_test, y_test_pred_1)
test_accuracy = accuracy_score(y_test, y_test_pred_1)
print(f"Test Accuracy: {test_accuracy:.4f}")
print(cm)
print(classification_report(y_test, y_test_pred_1))


# Plotting the confusion matrix
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues',
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()

# Plot AUROC and PR curves on test set with 95% confidence intervals
plot_roc_pr_curves(y_test, y_test_probs_1)
```

3.3 UCI Adult PCA+KNN Code

```
import sys
sys.path.append('../')
from imports import *

# loading processed data
df_train = pd.read_csv('./uci-adult-processed-data/train.csv')
df_val = pd.read_csv('./uci-adult-processed-data/val.csv')
df_test = pd.read_csv('./uci-adult-processed-data/test.csv')

# Define your features and target variable
target_column = 'target'
features = df_train.columns.drop(target_column)

# Separate features and target and convert to NumPy arrays
X_train = df_train[features].to_numpy()
y_train = df_train[target_column].to_numpy()
X_val = df_val[features].to_numpy()
y_val = df_val[target_column].to_numpy()
X_test = df_test[features].to_numpy()
y_test = df_test[target_column].to_numpy()

# Initialize the StandardScaler
scaler = StandardScaler()
# Fit the scaler on the training data and transform the training data
X_train = scaler.fit_transform(X_train)

# Transform the validation and test data
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
```

```
# Step 1: Apply PCA
# Choose the number of components, e.g., 2 for visualization
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)
X_val_pca = pca.transform(X_val)
X_test_pca = pca.transform(X_test)

# Step 2: Train KNN Classifier
# Choose the number of neighbors, e.g., 3
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train_pca, y_train)

# Step 3: Predict and Evaluate the model
y_val_pred = knn.predict(X_val_pca)
val_accuracy = accuracy_score(y_val, y_val_pred)

print(f"Validation Accuracy: {val_accuracy:.4f}")

print(confusion_matrix(y_val, y_val_pred))
print(classification_report(y_val, y_val_pred))

# Hyperparameters to try
n_components_options = [1, 3, 5, 10]  # PCA components
n_neighbors_options = [1, 5, 10, 15, 20]      # KNN neighbors

best_score = 0
best_params = {}

for n_components in n_components_options:
    # Apply PCA
    pca = PCA(n_components=n_components)
    X_train_pca = pca.fit_transform(X_train)
    X_val_pca = pca.transform(X_val)

    for n_neighbors in n_neighbors_options:
        # Apply KNN
        knn = KNeighborsClassifier(n_neighbors=n_neighbors)
        knn.fit(X_train_pca, y_train)

        # Evaluate the model on the validation set
        score = knn.score(X_val_pca, y_val)

        # Update best score and parameters if current score is better
        if score > best_score:
            best_score = score
            best_params = {'pca_n_components': n_components,
            'knn_n_neighbors': n_neighbors}

# Print the best parameters and the corresponding score
print("Best parameters:", best_params)
print("Best score:", best_score)

# Assuming you have your best parameters from the grid search
```

```
n_components, n_neighbors = 3, 15
best_params = {'pca_n_components': n_components, 'knn_n_neighbors': n_neighbors}
best_n_components = best_params['pca_n_components']
best_n_neighbors = best_params['knn_n_neighbors']

# Apply PCA with the best number of components
pca = PCA(n_components=best_n_components)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Train KNN with the best number of neighbors
knn = KNeighborsClassifier(n_neighbors=best_n_neighbors)
knn.fit(X_train, y_train)

# Evaluate the model on the test set
test_score = knn.score(X_test, y_test)

# Print the test score
print(f"Test score: {test_score:.4f}")

# Use the final KNN model to predict on the validation set
y_val_pred = knn.predict(X_val)

# Generating the confusion matrix
cm = confusion_matrix(y_val, y_val_pred)
val_accuracy = accuracy_score(y_val, y_val_pred)
print(f"Val Accuracy: {val_accuracy:.4f}")
print(cm)
print(classification_report(y_val, y_val_pred))

# Use the final KNN model to predict on the test set
y_test_pred = knn.predict(X_test)

# Generating the confusion matrix
cm = confusion_matrix(y_test, y_test_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test Accuracy: {test_accuracy:.4f}")
print(cm)
print(classification_report(y_test, y_test_pred))

# Plotting the confusion matrix
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues',
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()
```

3.4 UCI Adult FNN Code

```
#%%

#%%

#%%
import sys

import numpy as np

sys.path.append('../')
from imports import *
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import ParameterGrid
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import roc_curve, auc
import seaborn as sns


#%%
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
#%%
torch.cuda.is_available()
#%%
import pandas as pd


# loading processed data
df_train = pd.read_csv('./uci-adult-processed-data/train.csv')
df_val = pd.read_csv('./uci-adult-processed-data/val.csv')
df_test = pd.read_csv('./uci-adult-processed-data/test.csv')

# Define your features and target variable
target_column = 'target'
features = df_train.columns.drop(target_column)

# Separate features and target
X_train = df_train[features]
y_train = df_train[target_column]
X_val = df_val[features]
y_val = df_val[target_column]
X_test = df_test[features]
y_test = df_test[target_column]

# Initialize the StandardScaler
scaler = StandardScaler()
# scaler = MinMaxScaler()
# Fit the scaler on the training data and transform the training data
```

```
X_train = scaler.fit_transform(X_train)

# Transform the validation and test data
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)

# Convert to NumPy arrays (if needed)
y_train = y_train.to_numpy()
y_val = y_val.to_numpy()
y_test = y_test.to_numpy()

#%%

#%%
# Convert Numpy arrays to PyTorch tensors
X_train, y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dtype=
X_val, y_val = torch.tensor(X_val, dtype=torch.float32), torch.tensor(y_val, dtype=torch.fl
X_test, y_test = torch.tensor(X_test, dtype=torch.float32), torch.tensor(y_test, dtype=torc

# Create dataloaders
train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=True)
val_loader = DataLoader(TensorDataset(X_val, y_val), batch_size=32)
test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=1)
#%%

#%%

#%%

#%%
class FNN(nn.Module):
    def __init__(self, input_size):
        super(FNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(64, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.sigmoid(x)
        return x
#%%
model = FNN(X_train.shape[1]).to(device)
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)

#%%
def calculate_accuracy(y_pred, y_true):
    # Convert predictions to binary values (0 or 1)

    predicted = torch.round(y_pred).view(-1)
```

```
    y_true = y_true.squeeze()
    return (predicted == y_true).sum().float() / len(y_true)

train_losses, val_losses = [], []
train_accuracies, val_accuracies = [], []
# Training and validation for certain epochs
epochs = 50
for epoch in range(epochs):  # number of epochs
    model.train()
    total_loss, total_acc = 0, 0
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data).squeeze()
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        total_acc += calculate_accuracy(output, target)

    avg_train_loss = total_loss / len(train_loader)
    avg_train_acc = total_acc / len(train_loader)
    train_losses.append(avg_train_loss)
    train_accuracies.append(avg_train_acc)

    # Validation
    model.eval()
    val_loss, val_acc = 0, 0
    with torch.no_grad():
        for data, target in val_loader:
            data, target = data.to(device), target.to(device)
            output = model(data).squeeze()
            val_loss += criterion(output, target).item()
            val_acc += calculate_accuracy(output, target)

    avg_val_loss = val_loss / len(val_loader)
    avg_val_acc = val_acc / len(val_loader)
    val_losses.append(avg_val_loss)
    val_accuracies.append(avg_val_acc)
    print(f'Epoch {epoch+1}, Training Loss: {avg_train_loss:.4f}, Training Accuracy: {avg_


#%%

#%%
import matplotlib.pyplot as plt

# Plotting training and validation losses
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.title('Training & Validation Loss')
plt.xlabel('Epochs')
```

```
plt.ylabel('Loss')
plt.legend()
train_accuracies = [acc.cpu().numpy() for acc in train_accuracies]
val_accuracies = [acc.cpu().numpy() for acc in val_accuracies]

# Plotting training and validation accuracies
plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Training Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.title('Training & Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()

#%%
# Testing the model
model.eval()
test_loss = 0
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        output = model(data)
        output = output.squeeze(1)
        test_loss += criterion(output, target).item()

        # For accuracy calculation
        predicted = torch.round(output).view(-1)  # Assuming binary classification
        correct += (predicted == target).sum().item()
        total += target.size(0)

test_loss /= len(test_loader)
test_accuracy = 100 * correct / total
print(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2f}%')

#%%
# Now let us perform hyperparamter tuning to find optimal parameters and replace them where
# Defining the parametr grid for hyperparameter tuning
param_grid = {
    'lr': [0.0001, 0.001, 0.01],
    'batch_size': [32, 64],
    'epochs': [15, 50]
}
#%%
def calculate_accuracy(y_pred, y_true):
    # Convert predictions to binary values (0 or 1)
    y_pred, y_true = y_pred.squeeze(), y_true.squeeze()
    predicted = torch.round(y_pred).view(-1)
    return (predicted == y_true).sum().float() / len(y_true)
#%%
def train_and_validate(model, train_loader, val_loader, criterion, optimizer, epochs):
```

```
    train_losses, val_losses = [], []
    train_accuracies, val_accuracies = [], []

    for epoch in range(epochs):
        model.train()
        total_loss, total_acc = 0, 0
        for data, target in train_loader:
            data, target = data.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(data).squeeze()
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
            total_acc += calculate_accuracy(output, target)

        avg_train_loss = total_loss / len(train_loader)
        avg_train_acc = total_acc / len(train_loader)
        train_losses.append(avg_train_loss)
        train_accuracies.append(avg_train_acc)

        model.eval()
        val_loss, val_acc = 0, 0
        with torch.no_grad():
            for data, target in val_loader:
                data, target = data.to(device), target.to(device)
                output = model(data).squeeze()
                val_loss += criterion(output, target).item()
                val_acc += calculate_accuracy(output, target)

        avg_val_loss = val_loss / len(val_loader)
        avg_val_acc = val_acc / len(val_loader)
        val_losses.append(avg_val_loss)
        val_accuracies.append(avg_val_acc)

    return avg_val_loss, avg_val_acc

#%%
best_val_acc = 0
best_params = {}

for params in ParameterGrid(param_grid):
    train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=params['batch_siz
    val_loader = DataLoader(TensorDataset(X_val, y_val), batch_size=params['batch_size'])

    model = FNN(X_train.shape[1]).to(device)
    criterion = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=params['lr'])

    val_loss, val_acc = train_and_validate(model, train_loader, val_loader, criterion, opti

    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_params = params
```

```
    print(f"Params: {params}, Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_ac

print(f"Best Params: {best_params}, Best Validation Accuracy: {best_val_acc:.4f}%")

#%%

#%%
model = FNN(X_train.shape[1]).to(device)
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
# Convert Numpy arrays to PyTorch tensors
X_train, y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dtype=
X_val, y_val = torch.tensor(X_val, dtype=torch.float32), torch.tensor(y_val, dtype=torch.fl
X_test, y_test = torch.tensor(X_test, dtype=torch.float32), torch.tensor(y_test, dtype=torc

# Create dataloaders
train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=True)
val_loader = DataLoader(TensorDataset(X_val, y_val), batch_size=32)
test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=1)

def calculate_accuracy(y_pred, y_true):
    # Convert predictions to binary values (0 or 1)

    predicted = torch.round(y_pred).view(-1)
    y_true = y_true.squeeze()
    return (predicted == y_true).sum().float() / len(y_true)


train_losses, val_losses = [], []
train_accuracies, val_accuracies = [], []
# Training and validation for certain epochs
epochs = 15
for epoch in range(epochs):  # number of epochs
    model.train()
    total_loss, total_acc = 0, 0
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data).squeeze()
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        total_acc += calculate_accuracy(output, target)

    avg_train_loss = total_loss / len(train_loader)
    avg_train_acc = total_acc / len(train_loader)
    train_losses.append(avg_train_loss)
    train_accuracies.append(avg_train_acc)

    # Validation
    model.eval()
    val_loss, val_acc = 0, 0
```

```
    with torch.no_grad():
        for data, target in val_loader:
            data, target = data.to(device), target.to(device)
            output = model(data).squeeze()
            val_loss += criterion(output, target).item()
            val_acc += calculate_accuracy(output, target)

    avg_val_loss = val_loss / len(val_loader)
    avg_val_acc = val_acc / len(val_loader)
    val_losses.append(avg_val_loss)
    val_accuracies.append(avg_val_acc)
    print(
        f'Epoch {epoch + 1}, Training Loss: {avg_train_loss:.4f}, Training Accuracy: {avg_
```

```
#%%

# Plotting training and validation losses
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.title('Training & Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
train_accuracies = [acc.cpu().numpy() for acc in train_accuracies]
val_accuracies = [acc.cpu().numpy() for acc in val_accuracies]

# Plotting training and validation accuracies
plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Training Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.title('Training & Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

```
#%%
# Testing the model
model.eval()
test_loss = 0
correct = 0
total = 0
y_true = y_test
y_pred = []
y_scores = []

with torch.no_grad():
    for data, target in test_loader:
```

```
            data, target = data.to(device), target.to(device)
            output = model(data).squeeze(1)
            y_scores.extend(output.cpu().numpy().flatten())
            test_loss += criterion(output, target).item()

            # For accuracy calculation
            predicted = torch.round(output).view(-1)  # Assuming binary classification
            y_pred.append(predicted)

            correct += (predicted == target).sum().item()
            total += target.size(0)
fpr, tpr, _ = roc_curve(y_true, y_scores)
roc_auc = auc(fpr, tpr)
test_loss /= len(test_loader)
test_accuracy = 100 * correct / total
print(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2f}%')
#%%
y_true = np.array(y_true)
#%%
y_pred = [pred.to('cpu') for pred in y_pred]
#%%
y_pred = np.array(y_pred)
#%%
y_pred.shape
#%%

# Calculate the confusion matrix
cm = confusion_matrix(y_true, y_pred)
print(cm)
# Plot the confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='g', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

#%%
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic for Neural Network')
plt.legend(loc="lower right")
plt.show()
#%%
```

4.1 Fashion MNIST Logistic Regression Code

```
import sys
```

```
sys.path.append('../')
from imports import *
import cudf
import cupy as cp
import cuml
from cuml import LogisticRegression as logreg

def plot_roc_pr_curves(y_true, y_probs, n_bootstraps=1000):
    """Perform bootstrapping to calculate 95% confidence intervals for
    ROC and PR curves and plot them"""
    bootstrap_auroc_scores = []
    bootstrap_average_precision_scores = []

    for _ in range(n_bootstraps):
        # Bootstrap sample (with replacement)
        indices = resample(np.arange(len(y_true)), replace=True)
        y_true_boot = y_true[indices]
        y_probs_boot = y_probs[indices]

        # Compute metrics for bootstrap sample
        bootstrap_auroc_scores.append(roc_auc_score(y_true_boot, y_probs_boot))
        bootstrap_average_precision_scores.append(average_precision_score(
        y_true_boot, y_probs_boot))

    # Compute confidence intervals
    auroc_lower = np.percentile(bootstrap_auroc_scores, 2.5)
    auroc_upper = np.percentile(bootstrap_auroc_scores, 97.5)
    ap_lower = np.percentile(bootstrap_average_precision_scores, 2.5)
    ap_upper = np.percentile(bootstrap_average_precision_scores, 97.5)

    # Calculate original ROC and PR curves
    fpr, tpr, _ = roc_curve(y_true, y_probs)
    precision, recall, _ = precision_recall_curve(y_true, y_probs)
    auroc = roc_auc_score(y_true, y_probs)
    average_precision = average_precision_score(y_true, y_probs)

    # Plotting
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

    # ROC Curve
    ax1.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {auroc:.2f})')
    ax1.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    ax1.set_xlabel('False Positive Rate')
    ax1.set_ylabel('True Positive Rate')
    ax1.set_title('Receiver Operating Characteristic')
    ax1.legend(loc="lower right", title=f'95% CI: [{auroc_lower:.2f}, {auroc_upper:.2f}]')

    # Precision-Recall Curve
    ax2.plot(recall, precision, color='blue', lw=2, label=f'PR curve
    (area = {average_precision:.2f})')
    ax2.set_xlabel('Recall')
    ax2.set_ylabel('Precision')
    ax2.set_title('Precision-Recall Curve')
    ax2.legend(loc="lower left", title=f'95% CI: [{ap_lower:.2f}, {ap_upper:.2f}]')
```

```
    plt.show()

    # Return the confidence intervals
    return (np.round(auroc_lower,2), np.round(auroc_upper,2)),
    (np.round(ap_lower,2), np.round(ap_upper,2))


# Load the saved datasets
X_train = np.load('./fashion-mnist-processed-data/train_data.npy')
X_val = np.load('./fashion-mnist-processed-data/val_data.npy')
X_test = np.load('./fashion-mnist-processed-data/test_data.npy')

y_train = np.load('./fashion-mnist-processed-data/train_targets.npy')
y_val = np.load('./fashion-mnist-processed-data/val_targets.npy')
y_test = np.load('./fashion-mnist-processed-data/test_targets.npy')

# Reshape the training data
n_samples, height, width = X_train.shape
X_train = X_train.reshape((n_samples, height*width))

# Similarly, reshape the validation and test data
n_val_samples, height_val, width_val = X_val.shape
X_val = X_val.reshape((n_val_samples, height_val*width_val))

n_test_samples, height_test, width_test = X_test.shape
X_test = X_test.reshape((n_test_samples, height_test*width_test))

# convert to cupy arrays
X_train_cp = cp.array(X_train, dtype=cp.float32)
y_train_cp = cp.array(y_train, dtype=cp.float32)
X_val_cp = cp.array(X_val, dtype=cp.float32)
y_val_cp = cp.array(y_val, dtype=cp.float32)
X_test_cp = cp.array(X_test, dtype=cp.float32)
y_test_cp = cp.array(y_test, dtype=cp.float32)

X_cudf = cudf.DataFrame.from_pandas(pd.DataFrame(X_train))

# Initialize and fit the logistic regression model
log_reg = logreg(penalty='none', output_type='numpy', max_iter=10000)
log_reg.fit(X_train_cp, y_train_cp)

# Predict on validation set
y_val_pred = log_reg.predict(X_val)

# Predict probabilities
y_val_probs = log_reg.predict_proba(X_val)

# Evaluate the model
val_accuracy = accuracy_score(y_val, y_val_pred)
print(f"Validation Accuracy: {val_accuracy:.4f}")
print("Validation Classification Report:\n", classification_report(y_val, y_val_pred))

# Initialize and fit the logistic regression model
```

```
log_reg_l1 = logreg(penalty='l1', output_type='numpy', max_iter=10000)
log_reg_l1.fit(X_train_cp, y_train_cp)

# Predict on validation set
y_val_pred_l1 = log_reg_l1.predict(X_val)

# Predict probabilities
y_val_probs_l1 = log_reg_l1.predict_proba(X_val)

# Evaluate the model
val_accuracy = accuracy_score(y_val, y_val_pred_l1)
print(f"Validation Accuracy: {val_accuracy:.4f}")
print("Validation Classification Report:\n", classification_report(y_val, y_val_pred_l1))

# Initialize and fit the logistic regression model
log_reg_l2 = logreg(penalty='l2', output_type='numpy', tol=1e-2, max_iter=10000)
log_reg_l2.fit(X_train_cp, y_train_cp)

# Predict on validation set
y_val_pred_l2 = log_reg_l2.predict(X_val)

# Predict probabilities
y_val_probs_l2 = log_reg_l2.predict_proba(X_val)

# Evaluate the model
val_accuracy = accuracy_score(y_val, y_val_pred_l2)
print(f"Validation Accuracy: {val_accuracy:.4f}")
print("Validation Classification Report:\n", classification_report(y_val, y_val_pred_l2))

# Hyperparameter tuning
# Define a set of C values to try
C_values = [0.001, 0.01, 0.1, 1, 10, 100]

# Initialize variables to store the best score and corresponding C value
best_score = 0
best_C = None

# Perform grid search over the C values
for C in C_values:
    # Initialize and train the Logistic Regression model with L1 regularization
    log_reg_l1 = logreg(penalty='l1', C=C, output_type='numpy', tol=1e-2, max_iter=10000)
    log_reg_l1.fit(X_train_cp, y_train_cp)

    # Evaluate on the validation set
    y_val_pred_l1 = log_reg_l1.predict(X_val)
    score = accuracy_score(y_val, y_val_pred_l1)

    # If the score is better than the best score, update the best score and best C
    if score > best_score:
        best_score = score
        best_C = C

# Output the best C value
print(f"Best C value: {best_C} with AUROC: {best_score:.4f}")
```

```
# Final model: Logistic Regression with L1 regularization with C=0.01

# Train a new model using the best C value
best_log_reg_l1 = logreg(penalty='l1', C=best_C, output_type='numpy', max_iter=10000)
best_log_reg_l1.fit(X_train_cp, y_train_cp)

# Predict on validation set
y_val_pred_1 = best_log_reg_l1.predict(X_val)

# Predict probabilities
y_val_probs_1 = best_log_reg_l1.predict_proba(X_val)

# Evaluate the model
val_accuracy_1 = accuracy_score(y_val, y_val_pred_1)
print(f"Validation Accuracy: {val_accuracy_1:.4f}")
print("Validation Classification Report:\n", classification_report(y_val, y_val_pred_1))

# Predict on test set
y_test_pred_l1 = best_log_reg_l1.predict(X_test)

# Predict probabilities
y_test_probs_l1 = best_log_reg_l1.predict_proba(X_test)

# Evaluate the model
test_accuracy = accuracy_score(y_test, y_test_pred_l1)
print(f"Test Accuracy: {test_accuracy:.4f}")
print("Test Classification Report:\n", classification_report(y_test, y_test_pred_l1))
```

4.2 Fashion MNIST SVM Code

```
import sys
import os
sys.path.append('../')
from imports import *
import cudf
import cupy as cp
import cuml
from cuml.svm import SVC, LinearSVC
# from sklearn.svm import SVC

# Load the saved datasets
X_train = np.load('./fashion-mnist-processed-data/train_data.npy')
X_val = np.load('./fashion-mnist-processed-data/val_data.npy')
X_test = np.load('./fashion-mnist-processed-data/test_data.npy')

y_train = np.load('./fashion-mnist-processed-data/train_targets.npy')
y_val = np.load('./fashion-mnist-processed-data/val_targets.npy')
y_test = np.load('./fashion-mnist-processed-data/test_targets.npy')

# Reshape the training data
n_samples, height, width = X_train.shape
```

```
X_train = X_train.reshape((n_samples, height*width))

n_val_samples, height_val, width_val = X_val.shape
n_test_samples, height_test, width_test = X_test.shape

# Similarly, reshape the validation and test data if necessary
X_val = X_val.reshape((n_val_samples, height_val*width_val))
X_test = X_test.reshape((n_test_samples, height_test*width_test))

X_train_cp = cp.array(X_train, dtype=cp.float32)
y_train_cp = cp.array(y_train, dtype=cp.float32)

X_val_cp = cp.array(X_val, dtype=cp.float32)
y_val_cp = cp.array(y_val, dtype=cp.float32)

X_test_cp = cp.array(X_test, dtype=cp.float32)
y_test_cp = cp.array(y_test, dtype=cp.float32)

# Now, use X_train_subset and y_train_subset for training
# svm_linear = SVC(kernel='linear')
svm_linear = LinearSVC(loss='squared_hinge', penalty='l1', C=1,
probability=True, output_type='numpy')
svm_linear.fit(X_train_cp, y_train_cp)

# Predict and evaluate
y_val_pred = svm_linear.predict(X_val)
val_accuracy = accuracy_score(y_val, y_val_pred)

print(f"Validation Accuracy: {val_accuracy:.2f}")

print(confusion_matrix(y_val, y_val_pred))
print(classification_report(y_val, y_val_pred))

svm_poly = SVC(kernel='poly', probability=True, output_type='numpy')
svm_poly.fit(X_train_cp, y_train_cp)

# Predict and evaluate
y_val_poly_pred = svm_poly.predict(X_val)
val_accuracy = accuracy_score(y_val, y_val_pred)

print(f"Validation Accuracy SVM with polynomial kernel: {val_accuracy:.2f}")

print(confusion_matrix(y_val, y_val_poly_pred))
print(classification_report(y_val, y_val_poly_pred))

svm_rbf = SVC(kernel='rbf', probability=True, output_type='numpy')
svm_rbf.fit(X_train_cp, y_train_cp)

# Predict and evaluate
y_val_rbf_pred = svm_rbf.predict(X_val)
val_accuracy = accuracy_score(y_val, y_val_rbf_pred)

print(f"Validation Accuracy SVM with polynomial kernel: {val_accuracy:.2f}")
```

```
print(confusion_matrix(y_val, y_val_rbf_pred))
print(classification_report(y_val, y_val_rbf_pred))

# Predict and evaluate
y_test_rbf_pred = svm_rbf.predict(X_test)
test_accuracy = accuracy_score(y_test, y_test_rbf_pred)

print(f"Validation Accuracy SVM with polynomial kernel: {val_accuracy:.2f}")

print(confusion_matrix(y_test, y_test_rbf_pred))
print(classification_report(y_test, y_test_rbf_pred))
```

4.3 Fashion MNIST PCA+KNN Code

```
import sys
sys.path.append('../')
from imports import *

# Load the saved datasets
X_train = np.load('./fashion-mnist-processed-data/train_data.npy')
X_val = np.load('./fashion-mnist-processed-data/val_data.npy')
X_test = np.load('./fashion-mnist-processed-data/test_data.npy')

y_train = np.load('./fashion-mnist-processed-data/train_targets.npy')
y_val = np.load('./fashion-mnist-processed-data/val_targets.npy')
y_test = np.load('./fashion-mnist-processed-data/test_targets.npy')

# Reshape the training data
n_samples, height, width = X_train.shape
X_train_reshaped = X_train.reshape((n_samples, height*width))

n_val_samples, height_val, width_val = X_val.shape
n_test_samples, height_test, width_test = X_test.shape

# Similarly, reshape the validation and test data if necessary
X_val_reshaped = X_val.reshape((n_val_samples, height_val*width_val))
X_test_reshaped = X_test.reshape((n_test_samples, height_test*width_test))

X_train_reshaped.shape

x_train_1 = X_train.reshape((X_train.shape[0], -1))
x_train_1.shape

# Step 1: Apply PCA
# Choose the number of components, e.g., 2 for visualization
pca = PCA(n_components='mle')
X_train_pca = pca.fit_transform(X_train_reshaped)
X_val_pca = pca.transform(X_val_reshaped)
X_test_pca = pca.transform(X_test_reshaped)

# Step 2: Train KNN Classifier
# Choose the number of neighbors, e.g., 3
```

```
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train_pca, y_train)

# Step 3: Predict and Evaluate the model
y_val_pred = knn.predict(X_val_pca)
val_accuracy = accuracy_score(y_val, y_val_pred)

print(f"Validation Accuracy: {val_accuracy:.4f}")

print(confusion_matrix(y_val, y_val_pred))
print(classification_report(y_val, y_val_pred))

# Create a pipeline that first applies PCA and then KNN
pipe = Pipeline([
    ('pca', PCA()),
    ('knn', KNeighborsClassifier())
])

# Define the parameter grid to search
param_grid = {
    'pca__n_components': [0.2, 0.5, 0.9, 2, 5, 10, 50, 100],  # Example values
    'knn__n_neighbors': [1, 5, 10, 30, 50, 100]      # Example values
}

# Create a GridSearchCV object
grid_search = GridSearchCV(pipe, param_grid, cv=5, verbose=2)

# Fit the grid search to the data
grid_search.fit(X_train_reshaped, y_train)

# Print the best parameters and the corresponding score
print("Best parameters:", grid_search.best_params_)
print("Best cross-validation score:", grid_search.best_score_)

# now same pipeline with the best parameters
pca = PCA(n_components=100)
X_train_pca = pca.fit_transform(X_train_reshaped)
X_val_pca = pca.transform(X_val_reshaped)
X_test_pca = pca.transform(X_test_reshaped)

# Step 2: Train KNN Classifier
knn = KNeighborsClassifier(n_neighbors=10)
knn.fit(X_train_pca, y_train)

# Step 3: Predict and Evaluate the model
y_val_pred = knn.predict(X_val_pca)
val_accuracy = accuracy_score(y_val, y_val_pred)

print(f"Validation Accuracy: {val_accuracy:.4f}")

print(confusion_matrix(y_val, y_val_pred))
print(classification_report(y_val, y_val_pred))

# Step 4: Evaluate the model on test data
```

```
# Use the KNN model to predict on the test set
y_test_pred = knn.predict(X_test_pca)

# Generating the confusion matrix
cm = confusion_matrix(y_test, y_test_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test Accuracy: {test_accuracy:.4f}")
print(cm)
print(classification_report(y_test, y_test_pred))


# Plotting the confusion matrix
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()
```

4.4 Fashion MNIST 1-layer CNN Code

```
#%%
import sys
sys.path.append('../')
from imports import *
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import ParameterGrid
import concurrent.futures
from sklearn.metrics import confusion_matrix


#%%
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
#%%
torch.cuda.is_available()
#%%
# Load the saved datasets
train_data = np.load('./fashion-mnist-processed-data/train_data.npy')
val_data = np.load('./fashion-mnist-processed-data/val_data.npy')
test_data = np.load('./fashion-mnist-processed-data/test_data.npy')

train_targets = np.load('./fashion-mnist-processed-data/train_targets.npy')
val_targets = np.load('./fashion-mnist-processed-data/val_targets.npy')
test_targets = np.load('./fashion-mnist-processed-data/test_targets.npy')
#%%
train_data = np.expand_dims(train_data, axis=1)
val_data = np.expand_dims(val_data, axis=1)
test_data = np.expand_dims(test_data, axis=1)
```

```
#%%
# Converting to tensors
train_data, train_targets = torch.tensor(train_data, dtype=torch.float32),
torch.tensor(train_targets, dtype=torch.long)
val_data, val_targets = torch.tensor(val_data, dtype=torch.float32),
torch.tensor(val_targets, dtype=torch.long)
test_data, test_targets = torch.tensor(test_data, dtype=torch.float32),
torch.tensor(test_targets, dtype=torch.long)

#%%
# Normalizeing the data
train_data = train_data/ 255.0
val_data = val_data/255.0
test_data = test_data/255.0
#%%
# Data loaders and set batch size for training, validation, testing
batch_size = 64
train_loader = DataLoader(TensorDataset(train_data, train_targets),
batch_size=batch_size, shuffle=True)
val_loader = DataLoader(TensorDataset(val_data, val_targets),
batch_size=batch_size, shuffle=False)
test_loader = DataLoader(TensorDataset(test_data, test_targets),
batch_size=batch_size, shuffle=False)

#%%
#Single layer CNN model
class SingleLayerCNN(nn.Module):
    def __init__(self):
        super(SingleLayerCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.fc = nn.Linear(32 * 28 * 28, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

#%%

model = SingleLayerCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

#%%
# Training and validation for certain epochs
epochs = 10
for epoch in range(epochs):
    model.train()
    for i, (inputs, labels) in enumerate(train_loader):
        inputs, labels = inputs.to(device), labels.to(device)

        outputs = model(inputs)
        loss = criterion(outputs, labels)
```

```
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Validation phase
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        total_train = 0
        correct_train = 0

        for images_train, labels_train in train_loader:
            images_train, labels_train = images_train.to(device), labels_train.to(device)
            outputs_train = model(images_train)
            _, predicted_train = torch.max(outputs_train.data, 1)
            total_train += labels_train.size(0)
            correct_train += (predicted_train == labels_train).sum().item()
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_accuracy = 100 * correct_train / total_train
        val_accuracy = 100 * correct / total
        print(f'Epoch [{epoch+1}/{epochs}], Training accuracy:
        {train_accuracy:.2f}%, Validation Accuracy: {val_accuracy:.2f}%')

#%%
# Prediction on a single item from test set
model.eval()
with torch.no_grad():
    images, labels = next(iter(test_loader))
    images, labels = images.to(device), labels.to(device)
    outputs = model(images)
    _, predicted = torch.max(outputs.data, 1)
    print(f'Predicted label: {predicted[0].item()}, Actual label: {labels[0].item()}')
#%%
# Evaluation on test data
model.eval()
with torch.no_grad():
    correct_test = 0
    total_test = 0
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total_test += labels.size(0)
        correct_test += (predicted == labels).sum().item()

    test_accuracy = 100 * correct_test / total_test
    print(f'Test Accuracy: {test_accuracy:.2f}%')
```

```
#%%
# Now let us perform hyperparamter tuning to find optimal parameters
and replace them where needed!
# Defining the parametr grid for hyperparameter tuning
param_grid = {
    'lr': [0.001, 0.01],
    'batch_size': [32, 64],
    'epochs': [50, 100]
}
#%%
# Function to train and evaluate the model
def train_val_evaluate(model, train_loader, val_loader, criterion, optimizer, epochs=3):

    for epoch in range(epochs):
        model.train()
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

    model.eval()
    correct_val = 0
    total_val = 0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total_val += labels.size(0)
            correct_val += (predicted == labels).sum().item()

    val_accuracy = 100 * correct_val / total_val
    return val_accuracy
#%%
# Let us begin Hyperparameter tuning
best_accuracy = 0
best_params = None

for params in ParameterGrid(param_grid):
    # Update batch size for data loaders
    train_loader = DataLoader(TensorDataset(train_data, train_targets),
    batch_size=params['batch_size'], shuffle=True)
    val_loader = DataLoader(TensorDataset(val_data, val_targets),
    batch_size=params['batch_size'], shuffle=False)
    model = SingleLayerCNN().to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=params['lr'])

    # Training and evaluating the model
```

```
    accuracy = train_val_evaluate(model, train_loader, val_loader,
    criterion, optimizer, params['epochs'])

    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_params = params

    print(f"Params: {params}, Validation Accuracy: {accuracy:.2f}%")

print(f"Best Params: {best_params}, Best Accuracy: {best_accuracy:.2f}%")
#%%

model = SingleLayerCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

#%%

#%%

#%%
#Single layer CNN model
class SingleLayerCNN(nn.Module):
    def __init__(self):
        super(SingleLayerCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.fc = nn.Linear(32 * 28 * 28, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x


model = SingleLayerCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Initialize lists for accuracies and losses
train_accuracies = []
val_accuracies = []
train_losses = []
val_losses = []

epochs = 100
for epoch in range(epochs):
    model.train()
    total_train_loss = 0
    total_train = 0
    correct_train = 0

    for i, (inputs, labels) in enumerate(train_loader):
        inputs, labels = inputs.to(device), labels.to(device)
```

```
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        total_train_loss += loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        _, predicted_train = torch.max(outputs.data, 1)
        total_train += labels.size(0)
        correct_train += (predicted_train == labels).sum().item()

    avg_train_loss = total_train_loss / len(train_loader)
    train_losses.append(avg_train_loss)
    train_accuracy = 100 * correct_train / total_train
    train_accuracies.append(train_accuracy)

    # Validation phase
    model.eval()
    total_val_loss = 0
    correct_val = 0
    total_val = 0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            total_val_loss += loss.item()

            _, predicted = torch.max(outputs.data, 1)
            total_val += labels.size(0)
            correct_val += (predicted == labels).sum().item()

    avg_val_loss = total_val_loss / len(val_loader)
    val_losses.append(avg_val_loss)
    val_accuracy = 100 * correct_val / total_val
    val_accuracies.append(val_accuracy)

    print(f'Epoch [{epoch+1}/{epochs}], Training Loss: {avg_train_loss:.4f},
    Training Accuracy: {train_accuracy:.2f}%, Validation Loss: {avg_val_loss:.4f},
    Validation Accuracy: {val_accuracy:.2f}%')



#%%
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(train_accuracies, label='Training Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

```
plt.grid(True)

# Plotting the learning curves for loss
plt.subplot(1, 2, 2)
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.show()
#%%
model.eval()
predlist = torch.zeros(0, dtype=torch.long, device='cpu')
lbllist = torch.zeros(0, dtype=torch.long, device='cpu')

with torch.no_grad():
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)

        predlist = torch.cat([predlist, preds.view(-1).cpu()])
        lbllist = torch.cat([lbllist, labels.view(-1).cpu()])
#%%
conf_mat = confusion_matrix(lbllist.numpy(), predlist.numpy())
fig, ax = plt.subplots(figsize=(10, 10))  # Adjust the size as needed
sns.heatmap(conf_mat, annot=True, fmt='d', ax=ax, cmap="Blues")
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()

#%%
```

4.5 Fashion MNIST 2-layer CNN Code

```
#%%
import sys
sys.path.append('../')
from imports import *
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import ParameterGrid
import concurrent.futures
from sklearn.metrics import confusion_matrix


#%%
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
print("Using device:", device)
#%%
torch.cuda.is_available()
#%%
# Load the saved datasets
train_data = np.load('./fashion-mnist-processed-data/train_data.npy')
val_data = np.load('./fashion-mnist-processed-data/val_data.npy')
test_data = np.load('./fashion-mnist-processed-data/test_data.npy')

train_targets = np.load('./fashion-mnist-processed-data/train_targets.npy')
val_targets = np.load('./fashion-mnist-processed-data/val_targets.npy')
test_targets = np.load('./fashion-mnist-processed-data/test_targets.npy')
#%%
train_data = np.expand_dims(train_data, axis=1)
val_data = np.expand_dims(val_data, axis=1)
test_data = np.expand_dims(test_data, axis=1)

#%%
# Converting to tensors
train_data, train_targets = torch.tensor(train_data,
dtype=torch.float32), torch.tensor(train_targets, dtype=torch.long)
val_data, val_targets = torch.tensor(val_data,
dtype=torch.float32), torch.tensor(val_targets, dtype=torch.long)
test_data, test_targets = torch.tensor(test_data,
dtype=torch.float32), torch.tensor(test_targets, dtype=torch.long)

#%%
# Normalizeing the data
train_data = train_data/ 255.0
val_data = val_data/255.0
test_data = test_data/255.0
#%%
# Data loaders and set batch size for training, validation, testing
batch_size = 64
train_loader = DataLoader(TensorDataset(train_data, train_targets), batch_size=batch_size,
val_loader = DataLoader(TensorDataset(val_data, val_targets),
batch_size=batch_size, shuffle=False)
test_loader = DataLoader(TensorDataset(test_data, test_targets),
batch_size=batch_size, shuffle=False)

#%%
class TwoLayerCNN(nn.Module):
    def __init__(self):
        super(TwoLayerCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # Calculate the flattened size after conv and pooling layers
        self.flattened_size = 64 * 7 * 7  # Adjusted size for 2 layers of pooling
        self.fc = nn.Linear(self.flattened_size, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = self.pool(x)
```

```
        x = torch.relu(self.conv2(x))
        x = self.pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

#%%

model = TwoLayerCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

#%%
# Training and validation for certain epochs
epochs = 10
for epoch in range(epochs):
    model.train()
    for i, (inputs, labels) in enumerate(train_loader):
        inputs, labels = inputs.to(device), labels.to(device)

        outputs = model(inputs)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Validation phase
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        total_train = 0
        correct_train = 0

        for images_train, labels_train in train_loader:
            images_train, labels_train = images_train.to(device),
            labels_train.to(device)
            outputs_train = model(images_train)
            _, predicted_train = torch.max(outputs_train.data, 1)
            total_train += labels_train.size(0)
            correct_train += (predicted_train == labels_train).sum().item()
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_accuracy = 100 * correct_train / total_train
        val_accuracy = 100 * correct / total
        print(f'Epoch [{epoch+1}/{epochs}], Training accuracy:
        {train_accuracy:.2f}%, Validation Accuracy: {val_accuracy:.2f}%')

#%%
```

```
# Prediction on a single item from test set
model.eval()
with torch.no_grad():
    images, labels = next(iter(test_loader))
    images, labels = images.to(device), labels.to(device)
    outputs = model(images)
    _, predicted = torch.max(outputs.data, 1)
    print(f'Predicted label: {predicted[0].item()}, Actual label:
    {labels[0].item()}')
#%%
# Evaluation on test data
model.eval()
with torch.no_grad():
    correct_test = 0
    total_test = 0
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total_test += labels.size(0)
        correct_test += (predicted == labels).sum().item()

    test_accuracy = 100 * correct_test / total_test
    print(f'Test Accuracy: {test_accuracy:.2f}%')

#%%
# Now let us perform hyperparamter tuning to find optimal
parameters and replace them where needed!
# Defining the parametr grid for hyperparameter tuning
param_grid = {
    'lr': [0.001, 0.01, 0.005],
    'batch_size': [32, 64],
    'epochs': [25, 50, 100]
}
#%%
# Function to train and evaluate the model
def train_val_evaluate(model, train_loader, val_loader,
criterion, optimizer, epochs=3):

    for epoch in range(epochs):
        model.train()
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

    model.eval()
    correct_val = 0
    total_val = 0
    with torch.no_grad():
        for images, labels in val_loader:
```

```
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total_val += labels.size(0)
            correct_val += (predicted == labels).sum().item()

    val_accuracy = 100 * correct_val / total_val
    return val_accuracy
#%%
# Let us begin Hyperparameter tuning
best_accuracy = 0
best_params = None

for params in ParameterGrid(param_grid):
    # Update batch size for data loaders
    train_loader = DataLoader(TensorDataset(train_data, train_targets),
    batch_size=params['batch_size'], shuffle=True)
    val_loader = DataLoader(TensorDataset(val_data, val_targets),
    batch_size=params['batch_size'], shuffle=False)
    model = TwoLayerCNN().to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=params['lr'])

    # Training and evaluating the model
    accuracy = train_val_evaluate(model, train_loader, val_loader,
    criterion, optimizer, params['epochs'])

    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_params = params

    print(f"Params: {params}, Validation Accuracy: {accuracy:.2f}%")

print(f"Best Params: {best_params}, Best Accuracy: {best_accuracy:.2f}%")
#%%
# Data loaders and set batch size for training, validation, testing
batch_size = 64
train_loader = DataLoader(TensorDataset(train_data, train_targets), batch_size=batch_size,
val_loader = DataLoader(TensorDataset(val_data, val_targets),
batch_size=batch_size, shuffle=False)
test_loader = DataLoader(TensorDataset(test_data, test_targets),
batch_size=batch_size, shuffle=False)

model = TwoLayerCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

#%%

#%%

#%%
```

```
model = TwoLayerCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Initialize lists for accuracies and losses
train_accuracies = []
val_accuracies = []
train_losses = []
val_losses = []

epochs = 50
for epoch in range(epochs):
    model.train()
    total_train_loss = 0
    total_train = 0
    correct_train = 0

    for i, (inputs, labels) in enumerate(train_loader):
        inputs, labels = inputs.to(device), labels.to(device)

        outputs = model(inputs)
        loss = criterion(outputs, labels)
        total_train_loss += loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        _, predicted_train = torch.max(outputs.data, 1)
        total_train += labels.size(0)
        correct_train += (predicted_train == labels).sum().item()

    avg_train_loss = total_train_loss / len(train_loader)
    train_losses.append(avg_train_loss)
    train_accuracy = 100 * correct_train / total_train
    train_accuracies.append(train_accuracy)

    # Validation phase
    model.eval()
    total_val_loss = 0
    correct_val = 0
    total_val = 0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            total_val_loss += loss.item()

            _, predicted = torch.max(outputs.data, 1)
            total_val += labels.size(0)
            correct_val += (predicted == labels).sum().item()
```

```
    avg_val_loss = total_val_loss / len(val_loader)
    val_losses.append(avg_val_loss)
    val_accuracy = 100 * correct_val / total_val
    val_accuracies.append(val_accuracy)

    print(f'Epoch [{epoch+1}/{epochs}], Training Loss: {avg_train_loss:.4f},
    Training Accuracy: {train_accuracy:.2f}%,
    Validation Loss: {avg_val_loss:.4f}, Validation Accuracy: {val_accuracy:.2f}%')



#%%
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(train_accuracies, label='Training Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

# Plotting the learning curves for loss
plt.subplot(1, 2, 2)
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.show()
#%%
model.eval()
predlist = torch.zeros(0, dtype=torch.long, device='cpu')
lbllist = torch.zeros(0, dtype=torch.long, device='cpu')

with torch.no_grad():
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)

        predlist = torch.cat([predlist, preds.view(-1).cpu()])
        lbllist = torch.cat([lbllist, labels.view(-1).cpu()])
#%%
conf_mat = confusion_matrix(lbllist.numpy(), predlist.numpy())
fig, ax = plt.subplots(figsize=(10, 10))  # Adjust the size as needed
sns.heatmap(conf_mat, annot=True, fmt='d', ax=ax, cmap="Blues")
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```