Evaluating and Comparing Machine Learning Models for Classification

Sarwan Shah, Taha Mahmood, Sheikh Abdul Majid, Shoaib Imran

Name	Program	ASU ID
Sarwan Shah	Master of Electrical Engineering	1230055790
Taha Mahmood (Lead)	Master of Data Science, Analytics, and Engineering	1231265648
Sheikh Abdul Majid	Master of Data Science, Analytics, and Engineering	1230871345
Shoaib Imran	PhD in Electrical Engineering	1226788695

TABLE I
GROUP MEMBER DETAILS

I. INTRODUCTION

A. Datasets

We utilize three distinct datasets: the Wisconsin Breast Cancer Dataset, the UCI Adult Dataset, and the Fashion MNIST Dataset. Fashion MNIST is a numerical dataset containing 70,000 grayscale images, each 28 x 28 pixels in size. It has ten distinct labels corresponding to various fashion items like shirts, shoes, etc [6]. The Wisconsin Breast Cancer Dataset is a categorical dataset containing 569 samples, each characterized by 30 features. It focuses on the binary classification of tumors as either benign or malignant [7]. The UCI Adult Dataset is also a categorical dataset. This dataset contains 48,842 samples, each described by 14 features, of which eight are categorical, and six are integer-based. This dataset classifies individuals based on whether their income exceeds $50,000 or is less than that amount. [8]

B. Model Classes

We are using four model classes: Logistic Regression, Support Vector Machine (SVM), K-Nearest Neighbours (KNN), and Neural Networks. The logistic regression model employs a logistic function coupled with the explosive cross-entropy loss to learn a decision boundary that divides the classes data points. The SVM operates by identifying the optimal hyperplane that maximizes the margin between two distinct classes, effectively separating the data points of each type [12] [13]. On the other hand, the KNN classifier functions by locating the k closest neighbors to a specific data point and then uses a majority voting system among these neighbors to categorize the point. Neural networks are non-linear models that process data through layers. Data enters via the input layer, is transformed into hidden layers via weighted connections, and culminates in an output layer that classifies the results into different categories. Table II further mentions the advantages associated with each model class. [14]

Algorithm	Description	Advantages
K-Nearest Neighbors	Majority class among k-nearest neighbors	Simple, easy to understand
Logistic Regression	Probability modeling using logistic function	Simple, interpretable, computationally efficient
Support Vector Machine	Hyperplane to maximize margin	Effective in high-dimensional spaces, versatile
Neural Networks	Interconnected nodes for complex models	Highly expressive, effective for large data sets

TABLE II
MACHINE LEARNING ALGORITHMS OVERVIEW

C. Group Contribution

Table III provides a comprehensive listing of the codes and write-ups authored by each team member for various models. The writeups were compiled together in person.

Name	Models Assigned
Sarwan Shah	PCA & Logistic Regression
Taha Mahmood	PCA & K-Nearest Neighbors Classifiers
Sheikh Abdul Majid	PCA & Support Vector Machine (With and Without Kernels)
Shoaib Imran	Feedforward Neural Networks & Convolutional Neural Networks

TABLE III
MODEL ASSIGNMENT DETAILS

II. Breast Cancer Wisconsin Dataset

Fig. 16 in Appendix I-A illustrates a flowchart that further details the mehtodologies for this dataset.

A. Logistic Regression

A grid search was employed to find the best classification model using binary logistic regression and cross-entropy loss. This was done by spanning across the hyperparameter ranges shown in table IV. This spanning helped ensure convergence to minima using the distinctive optimizers that were explored: SGD, GD, and mini-batch GD, and to avoid over-fitting cases. SGD & GD were trained in separate flows, and GD was lumped with mini-batch GD by keeping the batch size equal to the training set. 3-fold cross-validation (three due to small dataset size) was used to ensure the robustness of the tuned hyperparameters. It was ensured that data standardization was performed during k-fold, and under-sampling was also performed on the dataset as a pre-processing stage to keep classes balanced (to avoid bias towards one class), as there were challenges working with class weights using the Pytorch library. In each flow (SGD & GD), out of the ensemble of trained models, two were selected: one based on the highest accuracy and another based on maximizing the true positive rate (TPR). As a conscious choice, the best model was chosen as the one that maximized the TPR to minimize the risk of misclassifying true cancer cases. The results for the best model are shown in Fig 1 and table V. The Pytorch library was used for training, while Sci-kit learn methods helped with evaluation metrics and optimizations. Results for the other three models are in appendix I-A.1.

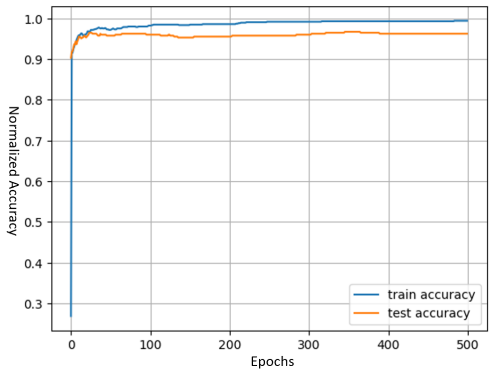

Fig. 1. Learning curve of best model.

Hyperparameter	**SGD**	GD/min-batch GD
Epochs	50,100,**500**,1000	50,100,500,1000
Learning Rate	**1**,0.1,0.01,0.001	1,0.1,0.01,0.001
Momentum/Batch Size	0,**0.5**,0.9,0.99	32,64,296
λ (l2 reg)	**0**,0.001,0.01,0.1,1	0, 0.001,0.01,0.1,1

TABLE IV

List of hyperparameters used for tuning model for SGD and GD.

Best model hyperparameters are bolded.

Metrics	*Optimizer	Acc	Opt.Acc	Opt.Threshold	TPR	FPR
Values	SGD	0.969	0.992	0.243	1	0

TABLE V

Performance metrics of best model.

B. Support Vector Machine

The dataset was trained both on the linear and non-linear Support Vector Machine (SVM) algorithms. The hyperparameters of interest include the Regularization parameter, C, for all the SVM models, with the addition of degree, d, in the Polynomial kernel and gamma, γ, in the Radial Basis Function (RBF) kernel. The range of the hyperparameters with reasoning has been listed in Appendix I-A.2. Using the Sci-Kit library [1], the grid search function was implemented to tune the hyperparameters due to its exhaustive search over the specified range, and as this dataset is tabular in nature with only five rows, so this method also performs well in terms of computational time. The accuracy for all three models was the same, as shown in Table VI. But, based on the learning curve of the linear SVM, it was concluded that it outperforms the other models in terms of accuracy when the number of training samples is increased, as shown in Appendix I-A.2. Figure 2 and Figure 3 deliver the linear SVM performance against the hyperparameter C and the number of training samples.

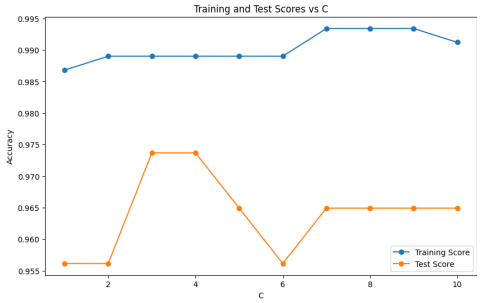

Fig. 2. Accuracy scores against C

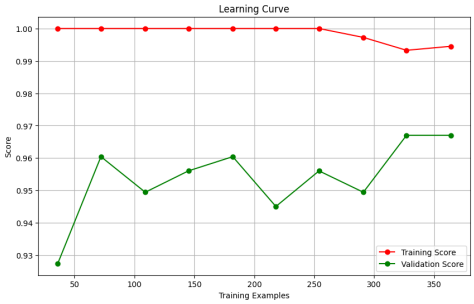

Fig. 3. Accuracy scores against samples

Kernel	Hyperparameters	Accuracy
Linear	C=3	0.970
Polynomial	C=6,d=1	0.970
RBF	C=6,γ=0.1	0.970

TABLE VI

Hyperparameters and accuracy of SVM models

C. k-Nearest Neighbors

The data set was trained using Sci-kit's KNN Classifier library [1], employing Grid Search with 10-fold cross-validation and a 20% train-test split on five essential hyperparameters. Before training, the dataset underwent dimensionality reduction through Principal Component Analysis (PCA), retaining 90% of its variance. The hyperparameters and their respective ranges are outlined in TABLE VII. [10] The third column of the TABLE VII named "Best (GS)" highlights the optimal parameters identified through Grid Search, along with the corresponding best model accuracy on the test data (refer to TABLE VIII). Furthermore, the Confusion Matrix of the model is mentioned in Fig 35. After Grid Search, the data set was trained iteratively across each hyperparameter range. This process aimed to generate hyperparameter tuning curves depicting the model's performance against the five hyperparameter ranges which is provided for Nearest Neighbors in Fig 4 and for other hyperparameter from Fig 30 to 33. [15]. The fourth column in TABLE VII named "Best (IS)" showcases the best hyperparameters derived iteratively, while the rationale behind each hyperparameter range mentioned in Appendix I-A.3.

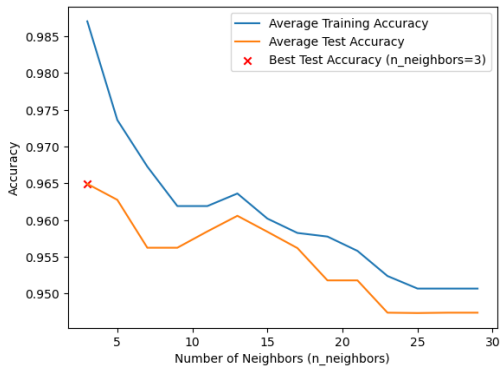

Fig. 4. Training and Test Accuracy against N Neighbours

Name	Range (GS)	Best (GS)	Best (IS)
No. of Neighbors	3, 5, 7, 9, 11, 13, 15 17, 19, 21, 23, 25, 27, 29	3	3
Weight Function	uniform, distance	distance	distance
Algorithm	auto, ball_tree, kd_tree, brute	brute	auto
Leaf Size	10, 20, 30, 40	30	10
P parameter	1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0	1.5	1.5

TABLE VII

RANGE OF HYPERPARAMETERS AND BEST HYPERPARAMETERS

Details/Metrics	Acc	F1	Precision	Recall
Values	0.939	0.939	0.939	0.939

TABLE VIII

PERFORMANCE METRICS FOR BEST MODEL

D. Neural Network

We used a three-layered, fully connected neural network (FCNN) with 175 neurons in each layer. We opted for a three-layered FCNN instead of a two-layered one, considering the relatively small size of this dataset, allowing us to enhance the model's complexity without substantially extending the training duration. We used Relu, dropout, and batch normalization after every linear layer. Fig. 38 in Appendix I-A.4 mentions the number of parameters for each layer in our network. The FCNN was trained using cross-entropy loss and Adam optimizer, utilizing the sci-kit-learn [1] and PyTorch [3] libraries. Table IX shows the range of hyperparameters over which the model was fine-tuned. The combination of hyperparameter values over which the final test accuracy was obtained is highlighted in bold. Table X presents the accuracy achieved on the test set. Fig. 37 in Appendix I-A.4 shows the confusion matrix obtained on the test set. Fig. 36 in Appendix I-A.4 shows the training and validation cross-entropy loss against the number of epochs. Moreover, Fig. 5 shows the training and validation accuracy plotted against the number of epochs.

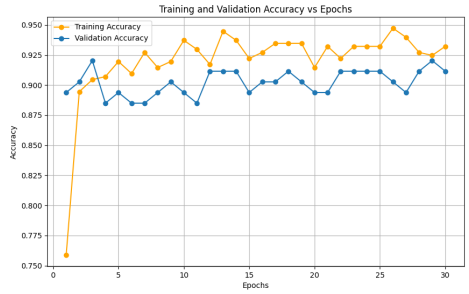

Fig. 5. Train and validation accuracy vs epochs

Hyperparameter	Adam
Epochs	10,20,**30**
Learning Rate	**0.001**, 0.01
Batch Size	16, **32**
λ (l2 reg)	**0.0001**, 0.01

TABLE IX

LIST OF HYPERPARAMETERS USED FOR TUNING THE MODEL

Metrics	Acc	F1	Precision	Recall
Values	0.9827	1.00	0.95	0.974

TABLE X

PERFORMANCE METRICS

E. Comparison of the Machine Learning Models

Logistic regression achieved the highest accuracy for this dataset. This superior performance of logistic regression may indicate the linear separability of features in this dataset.

III. Adult Income Dataset

This section describes the methodologies employed on the UCI adult income dataset. Fig. 42 in Appendix I-B illustrates a flowchart detailing our methodologies for this dataset.

A. Logistic Regression

A pipeline similar to the one discussed in Section II A was employed while learning the best classification model for the Adult Income dataset but with 5-fold cross-validation. However, due to limitations in computational capacity, the grid search could not be performed using classical gradient descent (GD) as an optimizer. Instead, only mini-batch GD was used at a fixed batch size of 1024 samples. Any fewer or more samples took too long to train. The ranges for other hyperparameters (shown in table XI) were also limited in the aforementioned for similar reasons. Moreover, since risk minimization was not a priority in this case, as it was for the breast cancer dataset, only one model was selected after the grid search in each flow (recall: separate flows for SGD & GD) such that it maximized the test accuracy after being optimized for the decision threshold using the ROC curve. The results for the best model are shown in fig 6 and table XII, while the results for the other model (based on SGD) are shown in appendix I-B.

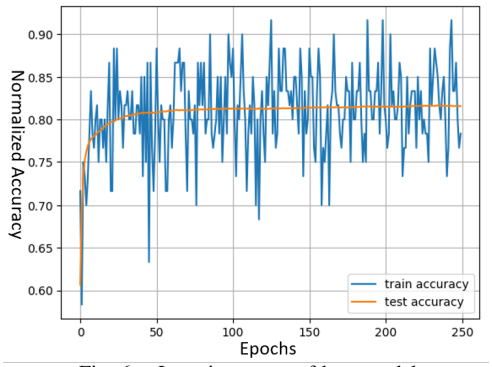

Fig. 6. Learning curve of best model.

Hyperparameter	SGD	**mini-batch GD**
Epochs	50,100,500,1000	50,100,**250**
Learning Rate	1,0.1,0.01,0.001	1,**0.5**,0.05
Momentum/Batch Size	0,0.5,0.9,0.99	**1024**
Lambda (l2 reg)	0,0.001,0.01,0.1,1	**0**,0.001,0.1

TABLE XI

List of hyperparameters used for tuning model for SGD and mini-batch GD. Best model hyperparameters are bolded.

Metrics	Optimizer	Acc	Opt.Acc	Opt.Threshold	TPR	FPR
Values	GD	0.8156	0.822	0.445	0.879	0.235

TABLE XII

Performance metrics of best model.

B. Support Vector Machine

The dataset was first trained on the linear and non-linear Support Vector Machine (SVM) algorithms. The hyperparameters of interest include the Regularization parameter, C, for all the SVM models, with the addition of degree, d, in the Polynomial kernel and gamma, γ, in the Radial Basis Function (RBF) kernel. The range of the hyperparameters, along with their selection reasoning, is mentioned in Appendix I-B.2. Due to the complexity of the dataset, first, it was compressed using Principal Component Analysis (PCA) with an Explained Variance of 75.96%. Then, a random search was implemented for the hyperparameter tuning due to its optimized search, which makes the code computationally efficient. The accuracy and selected hyperparameters for all three models have been listed in Table XIII. RBF SVM, being the most accurate, has been selected for the UCI dataset. Figure 7 and Figure 8 show the model performance of RBF SVM against γ and the number of training samples. A detailed comparison of all the SVM models has been given in Appendix I-B.2.

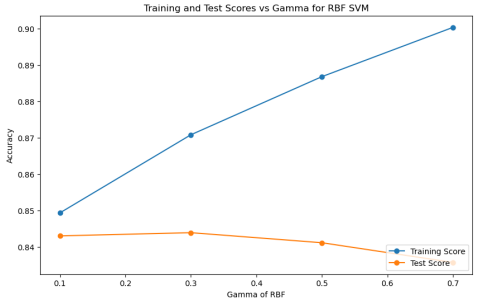

Fig. 7. Accuracy scores against γ

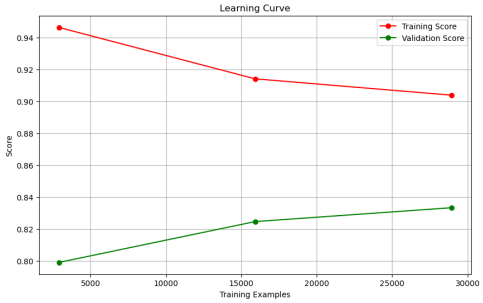

Fig. 8. Accuracy scores against samples

Kernal	Hyperparam	Accuracy
Linear	C=7	0.810
Polynomial	C=6,d=2	0.790
RBF	C=4,γ=0.3	0.840

TABLE XIII

Hyperparameters and accuracy of SVM models

C. k-Nearest Neighbors

Similarly to the Wisconsin data set, the UCI Adult data set was trained using Sci-kit's KNN Classifier [1] library, utilizing 10-fold Grid Search and a 20% train-test split on five hyperparameters. Before training, the dataset underwent dimensionality reduction through PCA to retain 90% variance. The hyperparameters and their ranges are outlined in TABLE XIV. The third column of TABLE XIV, labeled "Best (GS)," showcases the optimal parameters identified through Grid Search alongside the corresponding best model accuracy on the test data (refer to TABLE XV) and the Confusion Matrix in Appendix I-B.3 Fig 53. Following Grid Search, the dataset underwent individual training across each hyperparameter range, intending to generate hyperparameter tuning curves illustrating the model's performance across the five hyperparameter ranges. The fourth column in TABLE XIV, titled "Best (IS)," presents the best hyperparameters derived through this method, along with the associated tuning curves provided in Fig 9 for N Neighbors and the rest in Appendix I-B.3 Fig 48 to 51.

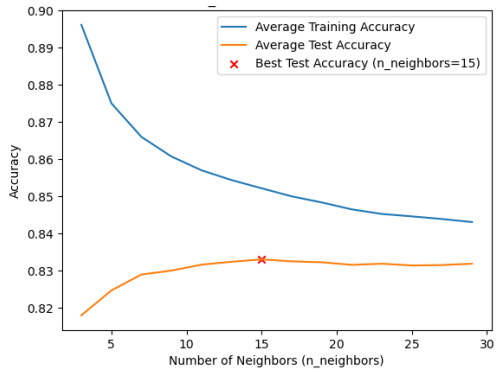

Fig. 9. Training and Test Accuracy against N Neighbours

Hyperparameters	Range (GS)	Best (GS)	Best (IS)
No. of Neighbors	3, 5, 7, 9, 11, 13, 15	13	15
Weight Function	uniform, distance	uniform	uniform
Algorithm	auto, ball_tree, kd_tree, brute	kd_tree	auto
Leaf Size	10, 20, 30, 40	10	20
P parameter	1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0	1.0	2.0

TABLE XIV
RANGE OF HYPERPARAMETERS AND BEST HYPERPARAMETERS

Metrics	Acc	F1	Precision	Recall
Values	0.832	0.826	0.824	0.832

TABLE XV
PERFORMANCE METRICS FOR BEST MODEL

D. Neural Network

We used a two-layered FCNN with 175 neurons in each layer. We used dropout, batch normalization, and Relu after every linear layer. Moreover, we used cross-entropy loss and Adam optimizer, utilizing the scikit-learn [1] and PyTorch [3] libraries. Table XVI shows the range of hyperparameters over which the model was fine-tuned. The combination of hyperparameter values over which the final test accuracy was obtained is highlighted in bold. Table XVII presents the accuracy achieved on the test set. Fig. 54 in Appendix I-B.4 shows the confusion matrix obtained. Fig. 55 in Appendix I-B.4 mentions the number of parameters for every layer in the neural network. Fig. 10 shows the training and validation accuracy plotted against the number of epochs. Fig. 56 in Appendix I-B.4 shows the training and validation loss plotted against the number of epochs.

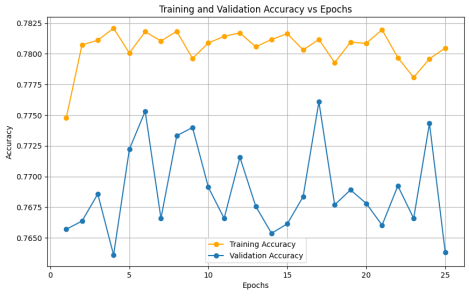

Fig. 10. Train and validation accuracy vs epochs

Hyperparameter	Adam
Epochs	**25**
Learning Rate	0.001, **0.01**
Batch Size	32, **64**
λ (l2 reg)	**0.0001**, 0.01

TABLE XVI
LIST OF HYPERPARAMETERS USED FOR TUNING THE MODEL

Metrics	Acc	F1	Precision	Recall
Values	0.774	0.153	0.989	0.083

TABLE XVII
PERFOMANCE METRICS

E. Comparison of the Machine Learning Models

We observe that the Support Vector Machine (SVM) with a Radial Basis Kernel (RBF) achieves the highest accuracy, 84%. In Appendix I-B.2, Table XXXI shows the classification report of the SVM RBF algorithm. Figure 47 shows the confusion matrix for the SVM RBF model. The second-highest accuracy was achieved by logistic regression, followed by the neural network. The superior performance of the SVM may be ascribed to its proficiency in handling non-linear decision boundaries and effectively managing high-dimensional feature spaces prevalent in the dataset.

IV. FashionMNIST Dataset

This section describes the methodologies employed on the Fashion MNIST dataset. Fig. 64 in Appendix I-C illustrates a flowchart that further details these methodologies.

A. Logistic Regression

We changed the logistic regression model from binary to multinomial as the dataset consisted of multiple classes. In addition, due to the high dimensionality of features in the data, principal component analysis (PCA) was performed on the features to reduce dimensionality and allow for quicker training and more accessible learning. The batch size for mini-batch GD was also limited to 4096 samples (as expressed in table IV), and k-fold cross-fold validation could not be performed as well due to computational limitations when working with this dataset. Instead, a simple train/test training loop was employed and coupled with the grid search to find the best model. Since this was a multinomial classification problem, the ROC curve could not be constructed, and the decision boundary was not optimized as in the case of the binary logistic regression explored in section II A & section III-A. The results for the best model are shown in fig 11 and table XIX. Results for the GD-based model can be found in appendix I-C.1

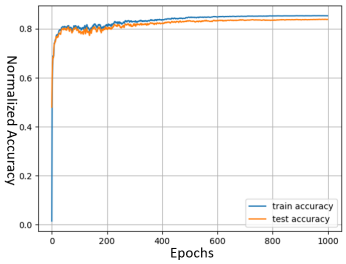

Fig. 11. Test accuracy of best model

Hyperparameter	SGD	**mini-batch GD**
Epochs	50,100,500,**1000**	50,100,250
Learning Rate	1,**0.1**,0.01,0.001	1,0.5,0.05
Momentum/Batch Size	0,0.5,0.9,**0.99**	4096
Lambda (l2 reg)	**0**,0.001,0.01,0.1,1	0,0.001,0.1

TABLE XVIII

List of hyperparams used for tuning model for SGD and mini-batch GD. Best model hyperparams are bolded.

Details/Metrics	Optimizer	Acc	Precision	Recall	F1
Values	SGD	0.8375	0.836	0.838	0.837

TABLE XIX

Performance metrics for best model.

B. Support Vector Machine

The dataset was trained both on the linear and non-linear Support Vector Machine (SVM) algorithms. The hyperparameters of interest include the Regularization parameter, C, for all the SVM models, with the addition of degree, d, in the Polynomial kernel and gamma, γ, in the Radial Basis Function (RBF) kernel. The range of the hyperparameters, along with their selection reasoning, is mentioned in Appendix I-C.2. Due to the dataset's complexity, it was first compressed using PCA with an Explained Variance of 70.07%. Then, a random search was implemented for the hyperparameter tuning due to its optimized search, which makes the code computationally efficient. The accuracy and selected hyperparameters for all three models have been listed in Table XX. Polynomial SVM, being the most accurate, has been selected for this dataset. Figure 12 and Figure 13 show the model performance of the Polynomial SVM against d and the number of training samples. A detailed comparison of all the SVM models has been given in Appendix I-C.2.

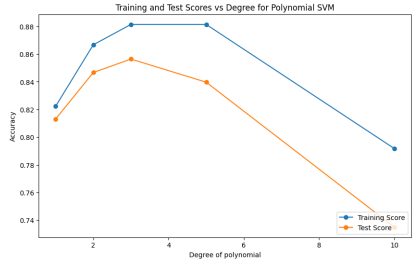

Fig. 12. Accuracy scores against d

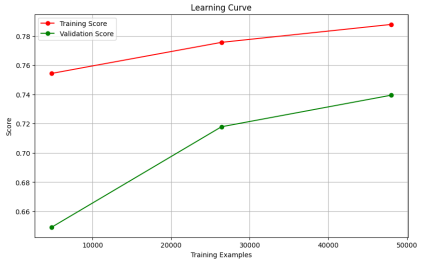

Fig. 13. Accuracy scores against samples

Kernal	Hyperparameters	Accuracy
Linear	C=9	0.810
Polynomial	C=7,d=5	0.840
RBF	C=6,γ=0.25	0.540

TABLE XX

Hyperparameters and accuracy of SVM models

C. k-Nearest Neighbors

The Fashion MNIST dataset underwent training using the Sci-kit's KNN Algorithm [1], wherein a dimensionality reduction to 90% of its variance was achieved through PCA. However, 10-fold CV GridSearch had to be replaced with 3-fold CV Random Search to account for the computational effort required to process the large dataset model. The investigation revealed the hyperparameter combination that yielded the highest accuracy on validation dataset which are highlighted in "Best (RS)" column in Table XXI [4], the confusion matrix on test dataset in Fig. 69 and the top 5 similar model in Table XXXIV. Furthermore, the hyperparameters were also search in an iterative manner which defined the hyperparameter tuning plots which can be found in Fig. 14 for the K Neighbors and the Appendix I-C.3 Fig 65 to 68 for the other 4 hyperparameters. The best hyperparameters from Iterative Search are highlighted in "Best (IS)" in Table XXI. [16]

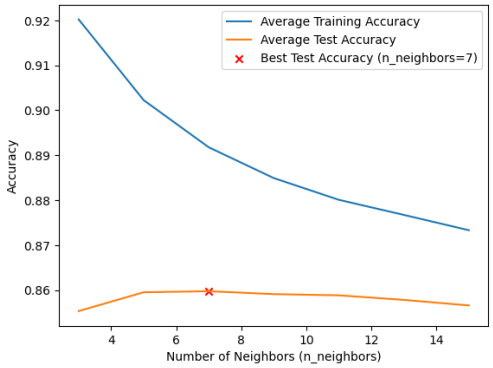

Fig. 14. Training and Test Accuracy against N Neighbours

Hyperparameters	Range (RS)	Best (RS)	Best (IS)
No. of Neighbors	3, 5, 7, 9, 11, 13, 15	7	7
Weight Function	uniform, distance	distance	distance
Algorithm	auto, ball_tree, kd_tree, brute	kd_tree	auto
Leaf Size	10, 20, 30, 40	30	10
P parameter	1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0	4.0	1.0

TABLE XXI

RANGE OF HYPERPARAMETERS AND BEST HYPERPARAMETERS

Metrics	Acc	F1	Precision	Recall
Values	0.867	0.868	0.869	0.867

TABLE XXII

PERFORMANCE METRICS FOR BEST MODEL

D. Neural Network

We utilized the convolutional neural network (CNN) architecture of LeNet5 [2] for this dataset. We used Adam optimizer and cross-entropy loss to train this neural network, utilizing the sci-kit-learn [1] and PyTorch [3] libraries. The range of values for hyperparameter tuning is detailed in Table XXIII. The set of hyperparameters that led to the optimal validation accuracy is emphasized in bold. These hyperparameters were chosen based on the 5-fold cross-validation accuracy obtained. Table XXIV outlines the model's accuracy on the test set, presenting accuracy. The confusion matrix, derived from the test data, is depicted in Fig.71 found in the Appendix I-C.4. Fig. 15 shows the training and validation accuracy plotted against the number of epochs. We observe that the training accuracy increases smoothly while the validation accuracy fluctuates. In addition, Fig. 70 in Appendix I-B.4 shows the training and validation cross-entropy loss plotted against the number of epochs.

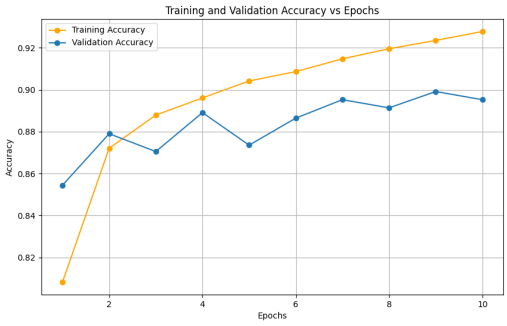

Fig. 15. Train and validation accuracy vs epochs

Hyperparameter	Adam
Epochs	5,**10**, 15
Learning Rate	**0.001**, 0.01
Batch Size	32, **64**

TABLE XXIII

LIST OF HYPERPARAMETERS USED FOR TUNING THE MODEL

Metrics	Accuracy
Values	0.8951

TABLE XXIV

PERFORMANCE METRICS

E. Comparison of the Machine Learning Models

We note that CNN achieves the best accuracy, followed by the support vector machine, aligning with our initial expectations. The CNN's superior accuracy can be credited to its ability to capture local connectivity within images, enabling it to learn hierarchical feature representations effectively. We also observed that CNN demonstrates lower computational complexity compared to SVM. This efficiency is primarily due to the shared parameters in the CNN's kernels, which additionally renders the network less susceptible to overfitting.

V. Value of Concepts Learned

A. Logistic Regression

The course's lectures were instrumental in elucidating why logistic regression achieves superior accuracy on the breast cancer dataset compared to other data sets. Especially the idea that logistic regression attempts to form a decision boundary by minimizing the error as its goal. In a sense, we are estimating the Bayesian Decision Rule (BDR), which is optimal in terms of error. This nature of the model was reflected in the breast cancer dataset on which it achieved the highest accuracy, thus also reflecting the high linear separability of the dataset. As we moved to data sets with seemingly higher feature dimensionality, it seemed to perform less well as models like the Neural Network or the SVM, which were making use of non-linear kernels. It is perhaps possible that if a non-linear kernel was used with logistic regression, it could have outperformed the SVM in terms of accuracy since the goal of SVM is to maximize margins rather than minimize error. Moreover, learning about the bias-variance trade-off provides insight into how the limited sample size of the Wisconsin dataset could have contributed to over-fitting in more complex models like neural networks.

B. Support Vector Machine

For the Support Vector Machine (SVM) algorithm, we applied a solid understanding of the mathematical foundations related to polynomial and RBF kernels, which we learned in the course. Moreover, the kernel trick allowed us to comprehend the non-linear mapping of the input features. We also utilized the cross-validation techniques for tuning the hyperparameters, which play an essential role in achieving optimization. Lastly, visualization techniques, like confusion matrix and learning curves, were also utilized to enhance our grasp of SVM intricacies.

C. k-Nearest Neighbours

The lectures have been instrumental in defining the intricacies of the k-Nearest Neighbors (KNN) algorithm and its classification methodology, particularly in handling vast amounts of data through an iterative process. Despite not achieving the highest accuracy in any data set, the insights gained from this course empowered us to skillfully explore numerous combinations of hyperparameters, ultimately leading us to discover optimal results.

D. Neural Network

The course's instruction on gradient descent provided valuable insights into various optimizers applicable to neural networks. We also acquired skills in determining the output image sizes based on kernel dimensions, which proved crucial in designing and troubleshooting our neural network models. Additionally, we learned that integrating non-linear functions after linear layers is essential for neural networks to capture the non-linearities present in datasets effectively.

VI. Conclusions

Our project entailed developing a comprehensive machine learning pipeline to process three distinct datasets: the Wisconsin Breast Cancer and UCI Adult datasets, binary classification problems, and the Fashion MNIST dataset, a multi-class classification challenge. We assessed the efficacy of four algorithms—Logistic Regression, K-Nearest Neighbor, Support Vector Machine, and Neural Network—to identify the most accurate model for each dataset. We achieved a commendable test accuracy on the Wisconsin dataset in comparison to the other two datasets. Notably, the Wisconsin dataset exclusively comprises float features, eliminating the need for preprocessing steps like encoding. However, the complex nature of the UCI Adult and Fashion MNIST datasets presented challenges in hyperparameter tuning due to extended execution times. To mitigate this, we employed Principal Component Analysis to reduce the dimensionality of the datasets in SVM, KNN, and Logistic Regression, thereby enhancing computational efficiency. Using the sklearn library, we implemented various functions, including grid search and random search cross-validation, to evaluate each model's accuracy and other performance metrics across the datasets. Our comprehensive analysis revealed that Logistic Regression achieved the highest accuracy for the Wisconsin Breast Cancer dataset. The Support Vector Machine with a Radial Basis Function Kernel emerged as the most accurate for the UCI Adult dataset. Finally, the Convolutional Neural Network was chosen for the Fashion MNIST dataset due to its superior accuracy.

REFERENCES

[1] Scikit-learn contributors. "scikit-learn: Machine Learning in Python." 2022. https://scikit-learn.org/stable/index.html

[2] "Blog. Paperspace. Writing LeNet5 from Scratch in PyTorch" 2022. https://blog.paperspace.com/writing-lenet5-from-scratch-in-python/

[3] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32 (pp. 8024–8035). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[4] Neeraj Kumar., and Li Zhang, and Shree K. Nayar. What is a Good Nearest Neighbors Algorithm for Finding Similar Patches in Images? The 10th European Conference on Computer Vision (ECCV)., October 2008 https://neerajkumar.org/projects/nnsearch/

[5] Swathi Nayak., Manisha Bhat1., N V Subba Reddy. and B Ashwath Rao. Study of distance metrics on k - nearest neighbor algorithm for star categorization. Citation Swathi Nayak et al 2022 J. Phys.: Conf. Ser. 2161 012004 DOI 10.1088/1742-6596/2161/1/012004

[6] Han Xiao and Kashif Rasul and Roland Vollgraf., Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. 2017-08-28. arXiv. cs.LG/1708.07747

[7] Wolberg,William, Mangasarian,Olvi, Street,Nick, and Street,W.. (1995). Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository. https://doi.org/10.24432/C5DW2B.

[8] Becker,Barry and Kohavi,Ronny. (1996). Adult. UCI Machine Learning Repository. https://doi.org/10.24432/C5XW20.

[9] Datascience stackexchange https://datascience.stackexchange.com/questions/76941/training-accuracy-greater-than-validation-accuracy

[10] Uddin, S., Haque, I., Lu, H. et al. Comparative performance analysis of K-nearest neighbour (KNN) algorithm and its different variants for disease prediction. Sci Rep 12, 6256 (2022). https://doi.org/10.1038/s41598-022-10358-x

[11] https://www.ibm.com/docs/en/spss-modeler/18.2.2?topic=node-svm-expert-options SVM Node Expert Options, 2021-03-02

[12] https://www.analyticsvidhya.com/blog/2021/10/support-vector-machinessvm-a-complete-guide-for-beginners/, Guide on Support Vector Machine (SVM) Algorithm, October 27th, 2023

[13] The Development and Application of Support Vector Machine Zhao Jun 2021 J. Phys.: Conf. Ser. 1748 052006

[14] Hastie, T., Tibshirani, R., Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd ed.). Stanford, CA: Stanford University. https://doi.org/10.1007/978-0-387-84858-7

[15] Murphy, Kevin P.. Machine learning : a probabilistic perspective. Cambridge, Mass. [u.a.]: MIT Press, 2013.

[16] D. L. Sankar, “EEE549 Statistical Machine Learning: From Theory to Practice,” 2023. [Online]. Available: Arizona State University

APPENDIX I
FURTHER DETAILS ON ALGORITHMS AND DATASET

A. Breast Cancer Wisconsin Dataset

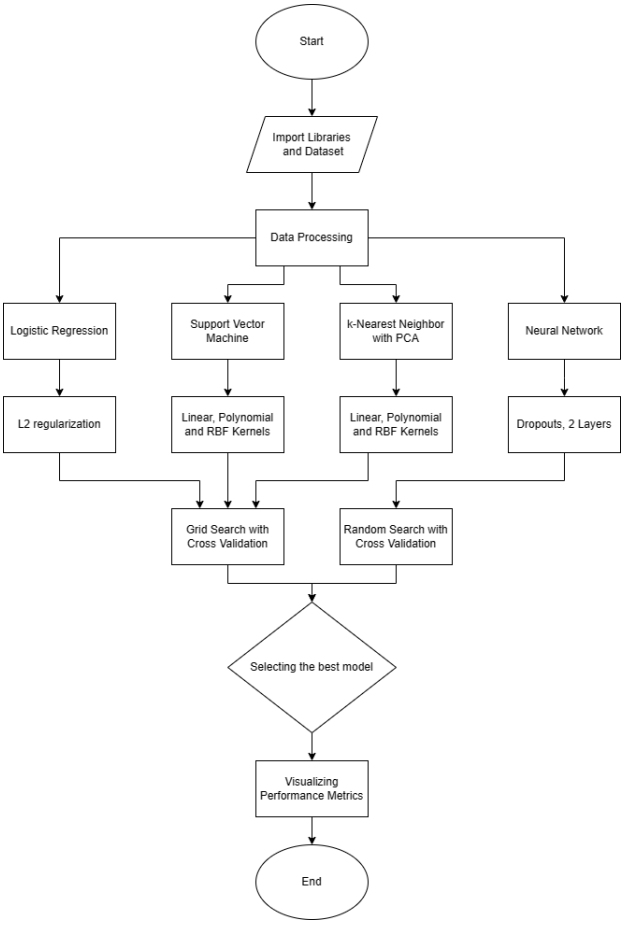

Fig. 16. High-level Training Process Flowchart For Breast Cancer Wisconsin Dataset

1) Logistic Regression:

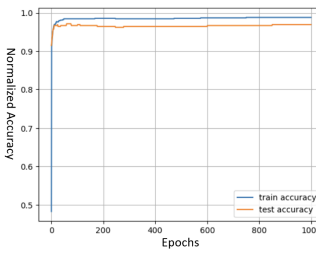

Fig. 17. Learning curve: top acc model (SGD)

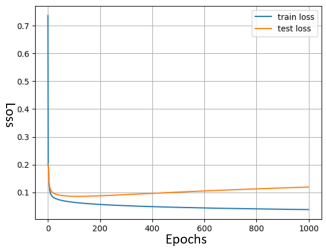

Fig. 18. Loss curve: top acc model (SGD)

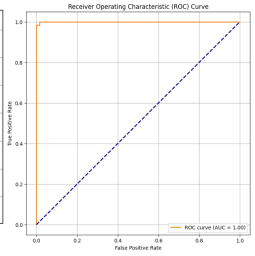

Fig. 19. ROC curve: top acc model (SGD)

Details	Values
Epochs	1000
Learning rate	1
Momentum	0.5
Lambda	0
Acc	0.967
Opt. Acc	0.992
Opt. Threshold	0.206
TPR	0.983
TPR	0.014

TABLE XXV
DETAILS OF TOP ACC MODEL (SGD)

2) Support Vector Machine: The range [1, 10] for the regularization parameter C allows for a balanced exploration of model complexity. Moreover, for polynomial SVM, the range [1, 30] allows thorough exploration, as low-degree polynomials might underfit, while high-degree polynomials could lead to overfitting. This range ensures consideration of a broad spectrum of polynomial features. Lastly, for the RBF SVM, the range [0.1, 2.0] strikes a balance, enabling exploration of both local and global patterns in the data without excessively emphasizing individual points. [11]

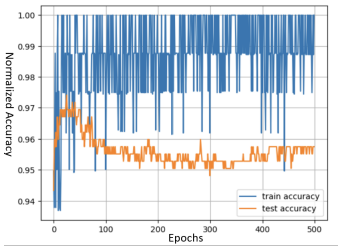

Fig. 20. Learning curve: top acc model (GD)

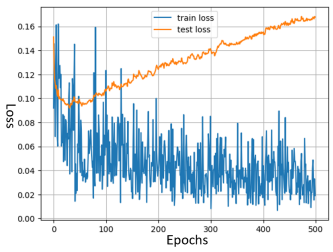

Fig. 21. Loss curve: top acc model (GD)

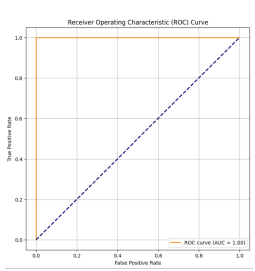

Fig. 22. ROC curve: top acc model (GD)

Details	Values
Epochs	500
Learning rate	1
Batch size	64
Lambda	0.001
Acc	0.957
Opt. Acc	0.992
Opt. Threshold	0.202
TPR	1
TPR	0

TABLE XXVI
DETAILS OF TOP ACC MODEL (GD)

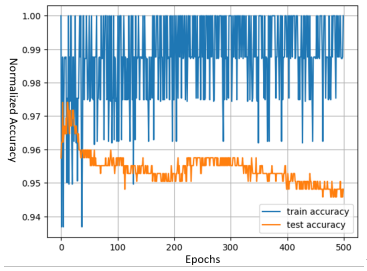

Fig. 23. Learning curve: top tpr model (GD)

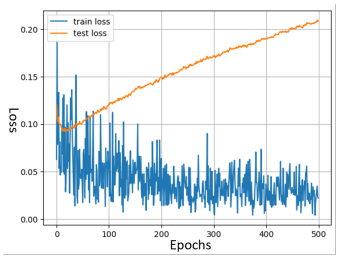

Fig. 24. Loss curve: top tpr model (GD)

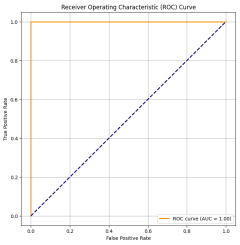

Fig. 25. ROC curve: top tpr model (GD)

Details	Values
Epochs	500
Learning rate	1
Batch size	32
Lambda	0
Acc	0.948
Opt. Acc	0.992
Opt. Threshold	0.273
TPR	1
TPR	0

TABLE XXVII
DETAILS OF TOP TPR MODEL (GD)

Hyperparameters	Linear	Polynomial
C	1-10	1-10
d		1-30
γ		

TABLE XXVIII
RANGE OF HYPERPARAMETERS IN SVM FOR WISCONSIN DATASET

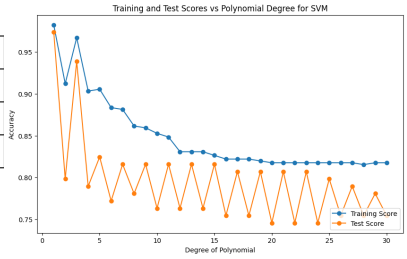

Fig. 26. Accuracy scores against d for Polynomial SVM

Fig. 27. Accuracy scores against γ for RBF SVM

3) k-Nearest Neighbor: The range of KNN Neighbors was selected between 3 and 29; there was a clear distinction between the M and B data sets; therefore, the low number of Neighbors is sufficient. Since custom weight functions yielded similar results, only two uniform and inverse distance weight functions were used. Since the quantum of processing required was low, a wide range of spatial algorithms have been used. Lastly, the ideal p range from 1.0 to 4.0 [5] is used since this accounts for the majority of Minkowski's decision complexity.

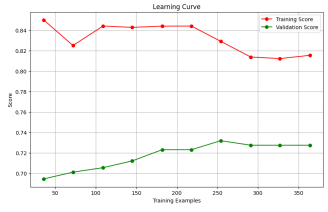

Fig. 28. Accuracy scores against samples for Polynomial SVM

Fig. 29. Accuracy scores against samples for RBF SVM

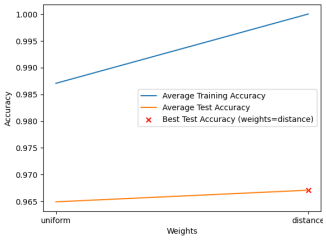

Fig. 30. Accuracy score against Weights

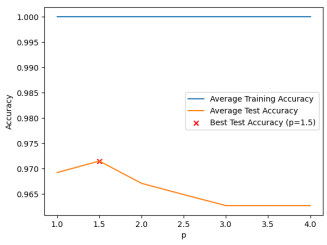

Fig. 31. Accuracy score against p

4) Neural Network:

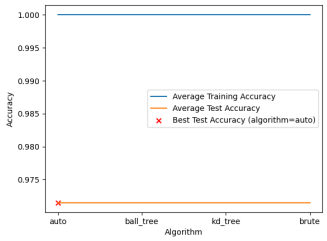

Fig. 32. Accuracy score against Spatial Algorithm

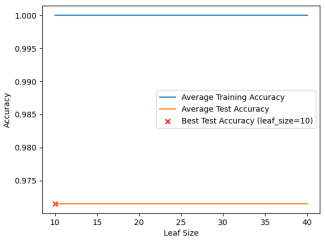

Fig. 33. Accuracy score against Leaf Size

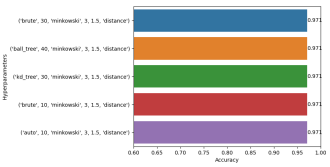

Fig. 34. Top 5 Model Hyperparameters using Grid Search

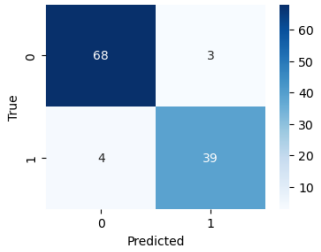

Fig. 35. Confusion Matrix of the Best Model

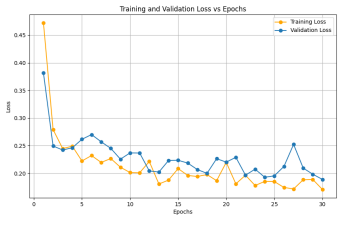

Fig. 36. Cross Entropy loss vs epochs for neural network on the Wisconsin dataset.

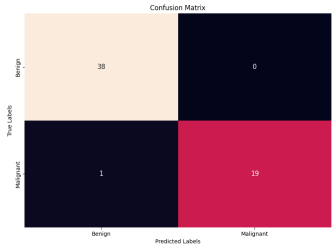

Fig. 37. Confusion Matrix from neural network on the Wisconsin dataset.

Layer (type)	Output Shape	Param #
Linear-1	[-1, 175]	5,425
BatchNorm1d-2	[-1, 175]	350
ReLU-3	[-1, 175]	0
Dropout-4	[-1, 175]	0
Linear-5	[-1, 175]	30,800
BatchNorm1d-6	[-1, 175]	350
ReLU-7	[-1, 175]	0
Dropout-8	[-1, 175]	0
Linear-9	[-1, 175]	30,800
BatchNorm1d-10	[-1, 175]	350
ReLU-11	[-1, 175]	0
Dropout-12	[-1, 175]	0
Linear-13	[-1, 2]	352
Total params:		68,427
Trainable params:		68,427

Fig. 38. Model summary of Neural Network used for Wisconsin dataset

B. Adult Income Dataset

1) Logistic Regression:

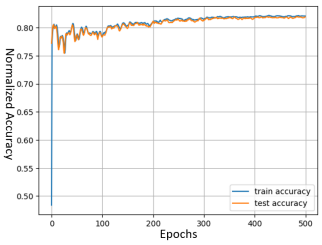

Fig. 39. Learning curve: top model (SGD)

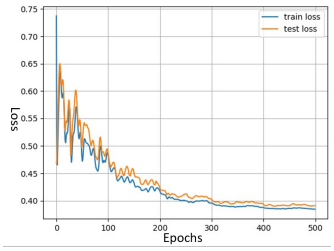

Fig. 40. Loss curve: top model (SGD)

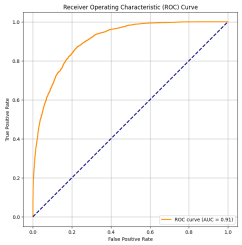

Fig. 41. ROC curve: top model (SGD)

Details	Values
Epochs	500
Learning rate	1
Momentum	0.99
Lambda	0.001
Acc	0.819
Opt. Acc	0.821
Opt. Threshold	0.476
TPR	0.856
FPR	0.213

TABLE XXIX
DETAILS OF TOP MODEL (SGD)

2) Support Vector Machine:

The range [1, 10] for the regularization parameter C allows for a balanced exploration of model complexity for linear SVM. However, for SVM with kernels, it was chosen to be [4, 6] in order to prevent a flexible decision boundary, and to speedup the hyperparameter tuning process. Moreover, for polynomial SVM, the chosen values for d, allow thorough exploration, as low-degree polynomials might underfit, while high-degree polynomials could lead to overfitting. Lastly, for the RBF SVM, the range was selected by 3/k and 6/k expressions, where k reprsents the number of features after conducting PCA. This strikes a balance, enabling exploration of both local and global patterns in the data without excessively emphasizing individual points. [12]

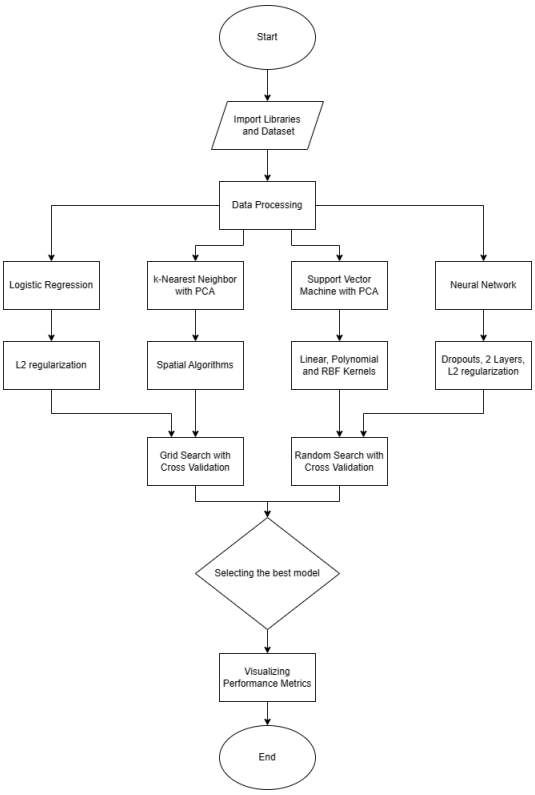

Fig. 42. High-level Training Process Flowchart For Adult Income Dataset

Hyperparameters	Linear	Polynomial	RBF
C	1-10	4-6	4-6
d		1, 2, 3, 5, 10	
γ			0.3, 0.4, 0.5, 0.6, 0.7

TABLE XXX
RANGE OF HYPERPARAMETERS IN SVM FOR WISCONSIN DATASET

3) k-Nearest Neighbor: The range of KNN Neighbors was selected between 3 and 15 for Grid Search; and 3 and 29 for Iterative Search to balance model robustness and sensitivity. Only two uniform and inverse distance weight functions were used while a wide Minkowski's p range from 1.0 to 4.0 [5] is used since this accounts for the majority of variation in model's performance. Since the dataset size was appropriate, four spatial algorithms have been used including brute algorithm which requires high computing for large datasets.

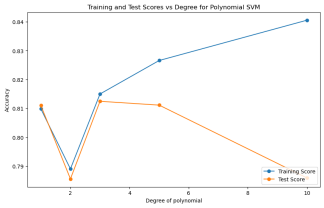

Fig. 43. Accuracy scores against d for Polynomial SVM

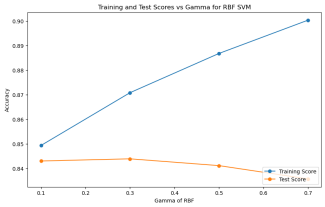

Fig. 44. Accuracy scores against γ for RBF SVM

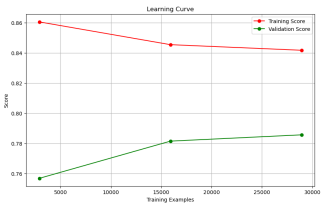

Fig. 45. Accuracy scores against samples for Polynomial SVM

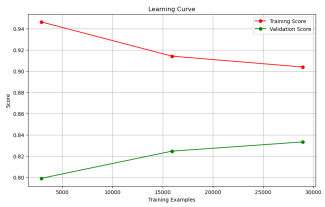

Fig. 46. Accuracy scores against samples for RBF SVM

4) Neural Network:

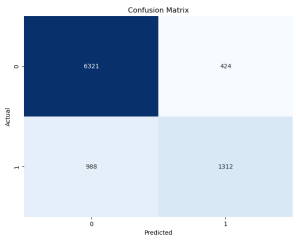

Fig. 47. Confusion Matrix for RBF SVM

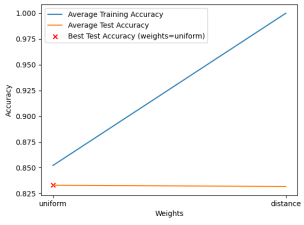

Fig. 48. Accuracy score against Weights

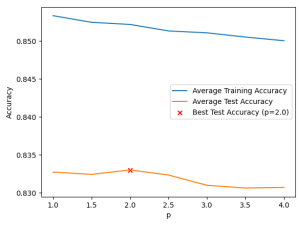

Fig. 49. Accuracy score against p

Class	Precision	Recall	F1 Score
0	0.86	0.94	0.90
1	0.76	0.57	0.65

TABLE XXXI
CLASSIFICATION REPORT

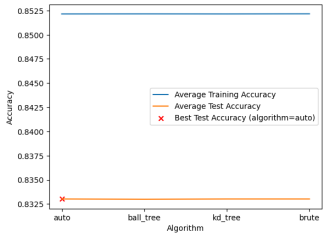

Fig. 50. Accuracy score against Spatial Algorithm

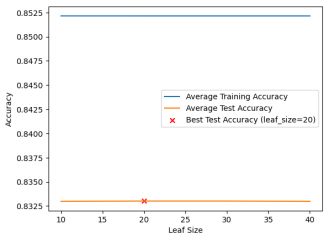

Fig. 51. Accuracy score against Leaf Size

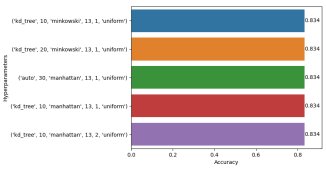

Fig. 52. Top 5 Model Hyper parameters using Grid Search

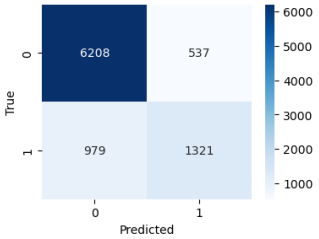

Fig. 53. Confusion Matrix of the Best Model

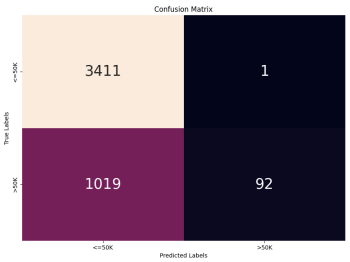

Fig. 54. Confusion matrix from neural network for the UCI Adult dataset

Layer (type)	Output Shape	Param #
Linear-1	[-1, 175]	2,625
BatchNorm1d-2	[-1, 175]	350
ReLU-3	[-1, 175]	0
Dropout-4	[-1, 175]	0
Linear-5	[-1, 175]	30,800
BatchNorm1d-6	[-1, 175]	350
ReLU-7	[-1, 175]	0
Dropout-8	[-1, 175]	0
Linear-9	[-1, 2]	352

Fig. 55. Summary of Neural Network for UCI dataset

Fig. 56. Cross-entropy loss vs epochs from Neural Network

C. FashionMNIST Dataset

1) Logistic Regression:

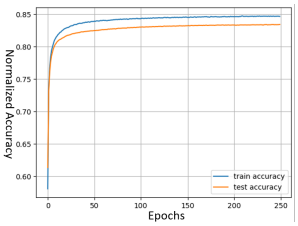

Fig. 57. Learning curve: top model (GD)

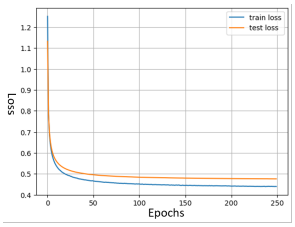

Fig. 58. Loss curve: top model (GD)

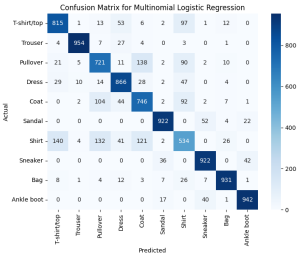

Fig. 59. C-Matrix: top model (GD)

Details	Values
Epochs	250
Learning rate	0.05
Batch Size	4096
Lambda	0.1
Acc	0.834
Precision	0.833
Recall	0.835
F1	0.834

TABLE XXXII
DETAILS OF TOP MODEL (GD)

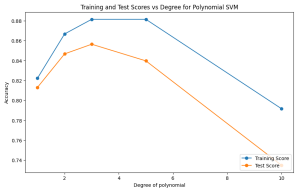

Fig. 60. Accuracy scores against d for Polynomial SVM

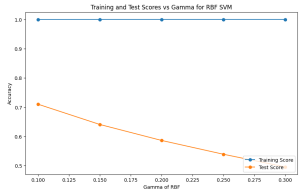

Fig. 61. Accuracy scores against γ for RBF SVM

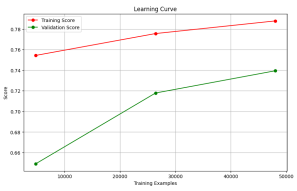

Fig. 62. Accuracy scores against samples for Polynomial SVM

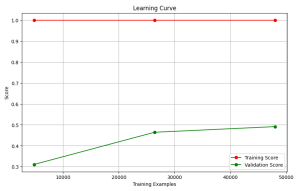

Fig. 63. Accuracy scores against samples for RBF SVM

2) Support Vector Machine: The range [1, 10] for the regularization parameter C allows for a balanced exploration of model complexity for linear SVM. However, for SVM with kernels, it was chosen to be [5, 7] in order to prevent a flexible decision boundary, and to speedup the hyperparameter tuning process. Moreover, for polynomial SVM, the sepecific values chosen for d, allows thorough exploration, as low-degree polynomials might underfit, while high-degree polynomials could lead to overfitting. Lastly, for the RBF SVM, the range was selected by 3/k and 6/k expressions, where k reprsents the number of features after conducting PCA. This strikes a balance, enabling exploration of both local and global patterns in the data without excessively emphasizing individual points.

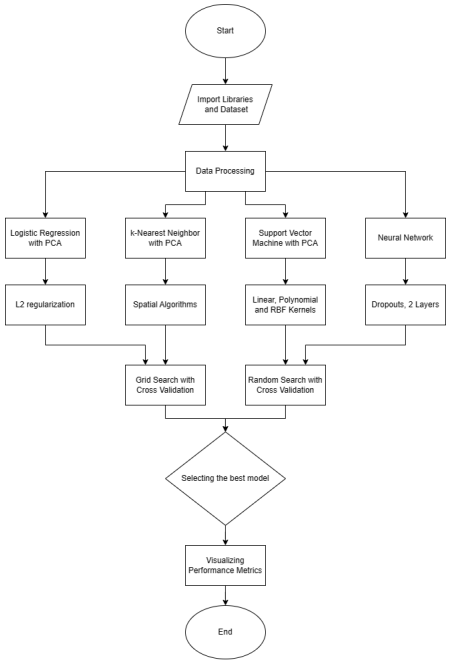

Fig. 64. High-level Training Process Flowchart For FashionMNIST Dataset

Hyperparameters	Linear	Polynomial	RBF
C	1-10	5-7	5-7
d		1, 2, 3, 5, 10	
γ			0.1, 0.15, 0.2, 0.25, 0.3

TABLE XXXIII
RANGE OF HYPERPARAMETERS IN SVM FOR WISCONSIN DATASET

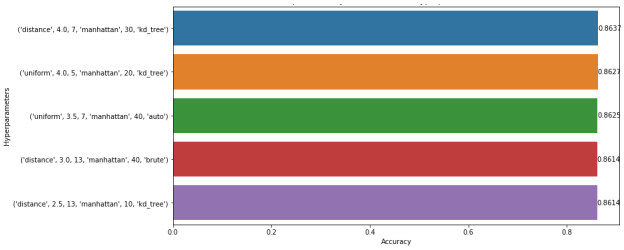

TABLE XXXIV
TOP 5 MODEL HYPER PARAMETERS USING RANDOM SEARCH

3) k-Nearest Neighbor: For Fashion MNIST, we have used a similar hyperparameters range as the UCI Adult Dataset. However, the range of the hyperparameter had to be reduced to account for the computational bandwidth. The range of KNN Neighbors was selected between 3 and 15 for Random Search and 3 and 29 for Iterative Search. Furthermore, two spatial algorithms were used for Iterative Search but extended to all four algorithms in the Random Search. The Minkowski's p range is set from 1.0 to 4.0, and the weighing matrix is kept uniform and distance, which is known to be the ideal range and yields appropriate results.

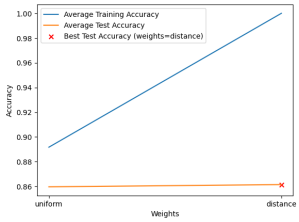

Fig. 65. Accuracy score against Weights

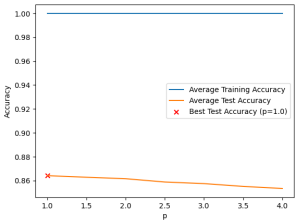

Fig. 66. Accuracy score against p

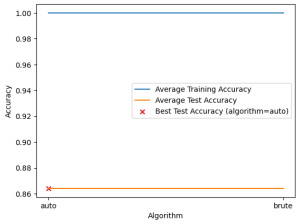

Fig. 67. Accuracy score against Spatial Algorithm

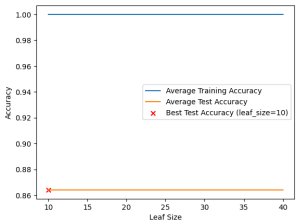

Fig. 68. Accuracy score against Leaf Size

4) Neural Network:

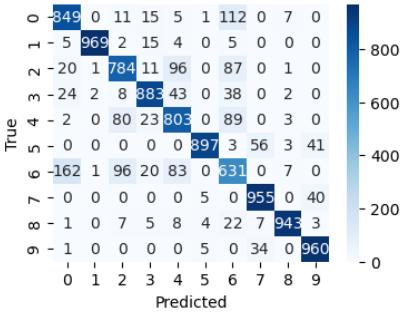

Fig. 69. Confusion Matrix of the Best Model for KNN

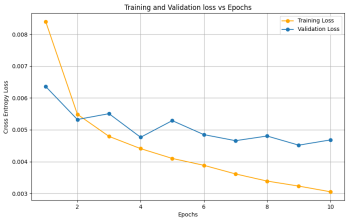

Fig. 70. Cross-entropy loss vs epochs using CNN for Fashion MNIST dataset.

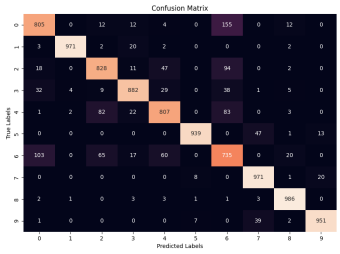

Fig. 71. Confusion matrix from the CNN for Fashion MNIST dataset.

APPENDIX II
CODE

A. Breast Cancer Wisconsin Dataset

1) Logistic Regression: Steps to run the code:

1) Run the file `SSHAH_TermProject_BreastCancerWisconsin_LogisticRegression_GD_Final.ipynb` in the directory Wisconsin/Logistic_Regression.
2) Run the file `SSHAH_TermProject_BreastCancerWisconsin_LogisticRegression_SGD_Final.ipynb` in the directory Wisconsin/Logistic_Regression.

```
# -*- coding: utf-8 -*-
"""SSHAH_TermProject_BreastCancerWisconsin_LogisticRegression_GD_Final.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/18un9eaEEDY8P9Yh5W9GF--N6Ku4ROKs3

## Importing Data and Libraries
"""

!pip install ucimlrepo

from ucimlrepo import fetch_ucirepo
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
from sklearn.decomposition import PCA
from sklearn.model_selection import KFold

import matplotlib.pyplot as plt
import seaborn as sns
import torch
import torch.nn as nn

import copy
import time

torch.manual_seed(42)
np.random.seed(42)

breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)

data = breast_cancer_wisconsin_diagnostic.data.features
data['labels'] = breast_cancer_wisconsin_diagnostic.data.targets

print(data.info())

"""## Pre-processing Data: Cleaning, Equalizing, and LabelEncoding"""

missing = data.isnull().sum()
duplicates = data.duplicated().sum()

data = data.dropna()
data = data.drop_duplicates()

print("Missing values removed:\n" + str(missing))
print("\nDuplicates removed:", duplicates)

# Balancing the data here to remove the class imbalance. Tried exploring class_weights to
address imbalance, but ran into issues with Pytorch.
# Hence used this approach as an alternative

classCount = data['labels'].value_counts()
minClassCount = classCount.min()
newData = pd.DataFrame(columns=data.columns)

for label in classCount.index:
      classSamples = data[data['labels'] == label]
      newSamples = classSamples.sample(minClassCount)
      newData = pd.concat([newData, newSamples])

data = newData
print(data.info())
```

```
# Encoding the labels with 0 and 1

featuresRaw = data.drop('labels', axis=1)
labelsRaw = data.labels

labelMapping = {}

labelEncoder = LabelEncoder()
labelsEncoded = labelEncoder.fit_transform(labelsRaw)

for class_ in labelEncoder.classes_:
  encoding = labelEncoder.transform([class_])[0]
  labelMapping[encoding] = class_

labels = labelsEncoded
print(labelMapping)

"""## Splitting Data"""

X_train, X_test, y_train, y_test = train_test_split(featuresRaw, labels, test_size=0.3,
random_state=42)

trainScaler = StandardScaler()
X_train = trainScaler.fit_transform(X_train)
X_test = trainScaler.transform(X_test)

X_train_df = pd.DataFrame(X_train, columns=featuresRaw.columns)
X_test_df = pd.DataFrame(X_test, columns=featuresRaw.columns)

X_train_df.head()

X_data = np.array(featuresRaw)
y_data = np.array(labels)

pca = PCA()
X_train_pca = pca.fit_transform(X_train)

varianceRatio = np.cumsum(pca.explained_variance_ratio_)
threshold = 0.95

principalVectors = np.argmax(varianceRatio >= threshold) + 1
X_train_reduced = X_train_pca[:, :principalVectors]

plt.plot(varianceRatio)
plt.xlabel('q - # of Principal Vectors')
plt.ylabel('Cumulative Explained Variance Ratio')
plt.title('Cumulative Explained Variance Ratio vs. # of Principal Vectors')
plt.grid(True)
plt.show()
print('# of principal vectors for 95% of the variance:', principalVectors)

"""## Feature Visualization"""

# Visualizing the spread of data-points of feature for the two classes. Interesting to observe
how they take a guassian shape, and
# gives an intutive sense of how a high-dimensional decision boundarys might separate them.

featuresLabelsDf = copy.deepcopy(X_train_df)
featuresLabelsDf['labels'] = copy.deepcopy(y_train)

col = len(featuresLabelsDf.columns) - 1
num_rows = (col - 1) // 4 + 1
num_cols = min(col, 4)

fig, axes = plt.subplots(nrows=num_rows, ncols=num_cols, figsize=(24, 22))
for i, column in enumerate(featuresLabelsDf.columns[:-1]):
```

```
    row = i // num_cols
    col = i % num_cols
    sns.histplot(featuresLabelsDf, x=column, hue='labels', kde=True, multiple='stack',
bins=30, ax=axes[row, col])
    axes[row,col].set_title('Distribution of ' + str(column) + ' by Class')
    axes[row,col].set_xlabel(column)
    axes[row,col].set_ylabel('Frequency')
    axes[row,col].legend(title='Class', labels=['Benign', 'Malignant'])

plt.tight_layout()
plt.show()

"""## Model Construction and Training"""

class LogisticRegression(nn.Module):
  def __init__(self, num_features, l2=0.0):
    super(LogisticRegression, self).__init__()
    self.layer1 = nn.Linear(num_features, 1)
    self.sig = nn.Sigmoid()
    self.l2 = l2


  def forward(self, x):
    out0 = self.layer1(x)
    out1 = self.sig(out0)
    return out1

# Helper Functions that perform training and reseting of model weights between iteration

def resetWeights(model):
    for layer in model.children():
        if isinstance(layer, nn.Linear):
            layer.reset_parameters()

def trainModelKFold(model, epochs, lossFn, X, y, lr=0.01, batchSize=32, decisionThreshold=0.5,
k=1):
    trainAcc, testAcc = np.zeros(epochs), np.zeros(epochs)
    trainErr, testErr = np.zeros(epochs), np.zeros(epochs)
    trainLoss, testLoss = np.zeros(epochs), np.zeros(epochs)

    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        trainScaler = StandardScaler()
        X_train = trainScaler.fit_transform(X_train)
        X_test = trainScaler.transform(X_test)

        X_train_tensor = torch.tensor(X_train).float()
        X_test_tensor = torch.tensor(X_test).float()
        y_train_tensor = torch.tensor(y_train).float()
        y_test_tensor = torch.tensor(y_test).float()

        resetWeights(model)
        train_dataset = torch.utils.data.TensorDataset(X_train_tensor, y_train_tensor)
        train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
batch_size=batchSize, shuffle=True)
        for epoch in range(epochs):
            for batch_X, batch_y in train_loader:
              output = model(batch_X)
              loss = lossFn(output.squeeze(), batch_y)
              loss.backward()

              with torch.no_grad():
                  for param in model.parameters():
```

```
                    param.data -= lr * param.grad

                model.zero_grad()

            with torch.no_grad():
                trainLoss[epoch] += loss.item()
                trainPredictions = (np.array(output.flatten()) > 0.5).astype(int)
                trainError = np.mean(trainPredictions != np.array(batch_y).astype(int))

                output = model(X_test_tensor)
                loss = lossFn(output.squeeze(), y_test_tensor)

                testLoss[epoch] += loss.item()
                testPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
                testError = np.mean(testPredictions != np.array(y_test_tensor).astype(int))

                trainErr[epoch] += trainError.item()
                testErr[epoch] += testError.item()
                trainAcc[epoch] += 1 - trainError.item()
                testAcc[epoch] += 1 - testError.item()

    avg_trainAcc = trainAcc/k
    avg_testAcc = testAcc/k
    avg_trainErr = trainErr/k
    avg_testErr = testErr/k
    avg_trainLoss = trainLoss/k
    avg_testLoss = testLoss/k

    return avg_trainAcc, avg_testAcc, avg_trainErr, avg_testErr, avg_trainLoss, avg_testLoss

def trainModel(model, epochs, lossFn, X_train, X_test, y_train, y_test, lr=0.01, batchSize=32,
decisionThreshold=0.5):
    trainAcc, testAcc = np.zeros(epochs), np.zeros(epochs)
    trainErr, testErr = np.zeros(epochs), np.zeros(epochs)
    trainLoss, testLoss = np.zeros(epochs), np.zeros(epochs)
    batchCount = 0

    resetWeights(model)
    train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
    train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batchSize,
shuffle=True)
    for epoch in range(epochs):
        for batch_X, batch_y in train_loader:
          batchCount += 1
          output = model(batch_X)
          loss = lossFn(output.squeeze(), batch_y)
          loss.backward()

          with torch.no_grad():
              for param in model.parameters():
                  param.data -= lr * param.grad

              model.zero_grad()

          with torch.no_grad():
              trainLoss[epoch] += loss.item()
              trainPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
              trainError = np.mean(trainPredictions != np.array(y_train).astype(int))

              output = model(X_test)
              loss = lossFn(output.squeeze(), y_test)

              testLoss[epoch] += loss.item()
              testPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
              testError = np.mean(testPredictions != np.array(y_test).astype(int))

              trainErr[epoch] += trainError.item()
```

```
                testErr[epoch] += testError.item()
                trainAcc[epoch] += 1 - trainError.item()
                testAcc[epoch] += 1 - testError.item()

    trainAcc = trainAcc/(batchCount/epochs)
    testAcc = testAcc/(batchCount/epochs)
    trainErr = trainErr/(batchCount/epochs)
    testErr = testErr/(batchCount/epochs)
    trainLoss = trainLoss/(batchCount/epochs)
    testLoss = testLoss/(batchCount/epochs)

    return trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss


X_train_tensor = torch.tensor(X_train).float()
y_train_tensor = torch.tensor(y_train).float()
X_test_tensor = torch.tensor(X_test).float()
y_test_tensor = torch.tensor(y_test).float()

"""## Grid Search Analysis"""

# Grid search for find the hyperparameter tuning to find best model

epochs_list = [50, 100, 500, 1000]
learning_rates = [1, 0.1, 0.01, 0.001]
batchSizes = [32, 64, len(X_train)]
lambdas = [0, 0.001, 0.01, 0.1, 1]
trainedModels = []

num_features = len(X_train_df.columns)
lossFn = nn.BCELoss()

for lambda_ in lambdas:
  model = LogisticRegression(num_features, l2=lambda_)
  for batchSize in batchSizes:
    for lr in learning_rates:
        for epochs in epochs_list:
          startTime = time.time()
          trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModelKFold(
              model,
              epochs,
              lossFn,
              X_data,
              y_data,
              batchSize=batchSize,
              lr=lr,
              k=3,
          )
          trainingTime = time.time() - startTime
          trainedModels.append({ 'model': copy.deepcopy(model),
                                'hyperparameters': {'epochs': epochs, 'lr': lr, 'batchSize':
batchSize, 'lambda': lambda_ },
                                'trainingStats': { 'testAcc': testAcc[-1], 'trainAcc':
trainAcc[-1],
                                                   'testError': testErr[-1], 'trainError':
trainErr[-1],
                                                   'testLoss': testLoss[-1], 'trainLoss':
trainLoss[-1] },
                                'trainingTime': trainingTime
                                })
          print(f'Completed epochs, lr, batchSize, lambda, test error: {epochs}, {lr},
{batchSize}, {lambda_}, --- {testAcc[-1]}')

"""## Find best model that maximizies accuracy using ROC curve:"""

def findOptimalThresholdForModelAcc(model, X_test_tensor, y_test_tensor):
  with torch.no_grad(): output = model(X_test_tensor)
```

```
  _, _, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
  optimalAcc = None
  optimalThreshold = None

  for threshold in thresholds:
    testPredictions = (np.array(output.detach().numpy().flatten()) > threshold).astype(int)
    testAcc = 1 - np.mean(testPredictions != np.array(y_test_tensor).astype(int))

    if (optimalThreshold == None):
      optimalThreshold = threshold
      optimalAcc = testAcc
    elif (testAcc > optimalAcc):
      optimalThreshold = threshold
      optimalAcc = testAcc



  return optimalAcc, optimalThreshold

globalOptimalModel = None
globalOptimalLoss = None
globalOptimalAcc = None
globalOptimalThreshold = None

for model_ in trainedModels:
    optimalAcc, optimalThreshold = findOptimalThresholdForModelAcc(
        model_['model'],
        X_test_tensor,
        y_test_tensor
    )
    optimalLoss = model_['trainingStats']['testLoss']

    if (globalOptimalLoss == None):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalAcc > globalOptimalAcc):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalAcc == globalOptimalAcc and optimalLoss < globalOptimalLoss):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold

print(globalOptimalModel)

gModel = globalOptimalModel['model']
with torch.no_grad(): output = gModel(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)

plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

```
gLr = globalOptimalModel['hyperparameters']['lr']
gEpochs = globalOptimalModel['hyperparameters']['epochs']
gBatchSize = globalOptimalModel['hyperparameters']['batchSize']
gLambda = globalOptimalModel['hyperparameters']['lambda']

index = int(np.where(thresholds == globalOptimalThreshold)[0])
FPR = fpr[index]
TPR = tpr[index]

print(f'Global Optimal Model Hyperparameters: lr={gLr}, epochs={gEpochs}, batchSize=
{gBatchSize}, lamda={gLambda}')
print(f'Global Optimal TPR: {TPR}, Global Optimal FPR: {FPR}, Global Optimal Threshold:
{globalOptimalThreshold}')
print(f'Global Optimal Acc: {globalOptimalAcc}, Global Optimal Loss: {globalOptimalLoss}')

"""### Reproducability Test & Visualization"""

lr = globalOptimalModel['hyperparameters']['lr']
epochs = globalOptimalModel['hyperparameters']['epochs']
batchSize = globalOptimalModel['hyperparameters']['batchSize']
lambda_ = globalOptimalModel['hyperparameters']['lambda']

num_features = len(X_train_df.columns)
model = LogisticRegression(num_features, l2=lambda_)
lossFn = nn.BCELoss()

trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModelKFold(
    model,
    epochs,
    lossFn,
    X_data,
    y_data,
    lr=lr,
    batchSize=batchSize,
    k=3,
)

model.eval()

plt.plot(trainAcc, label='train accuracy')
plt.plot(testAcc, label='test accuracy')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainErr, label='train error')
plt.plot(testErr, label='test error')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainLoss, label='train loss')
plt.plot(testLoss, label='test loss')
plt.grid(True)
plt.legend()

plt.show()


print(f'Train Accuracy: {trainAcc[-1]}, Train Error: {trainErr[-1]}, Train Loss: {trainLoss[-
1]}')
print(f'Test Accuracy: {testAcc[-1]}, Test Error: {testErr[-1]}, Test Loss: {testLoss[-1]}')

with torch.no_grad(): output = model(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)
```

```
plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

optimalAcc, optimalThreshold = findOptimalThresholdForModelAcc(
        model,
        X_test_tensor,
        y_test_tensor,
    )

index = int(np.where(thresholds == optimalThreshold)[0])
FPR = fpr[index]
TPR = tpr[index]

print(f'Global Optimal Model Hyperparameters: lr={lr}, epochs={epochs}, batchSize={batchSize},
lamda={lambda_}')
print(f'Global Optimal TPR: {TPR}, Global Optimal FPR: {FPR}, Global Optimal Threshold:
{optimalThreshold}')
print(f'Global Optimal Acc: {optimalAcc}, Global Optimal Loss: {testLoss[-1]}')

"""## Risk Minimization: Finding the Optimal Model such that it minimizes the risk by
increasing true positive rate (TPR)"""

def findOptimalThresholdForModel(model, X_test_tensor, y_test_tensor, fprThreshhold=0.25):
  with torch.no_grad(): output = model['model'](X_test_tensor).detach().numpy().flatten()
  fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(), output, pos_label=1)
  optimalTPR = None
  optimalFPR = None
  optimalThreshold = None

  for threshold in thresholds:
    index = int(np.where(thresholds == threshold)[0])
    FPR = fpr[index]
    TPR = tpr[index]

    if (optimalTPR == None):
      optimalThreshold = threshold
      optimalTPR = TPR
      optimalFPR = FPR
    elif (TPR > optimalTPR and FPR < fprThreshhold):
      optimalThreshold = threshold
      optimalTPR = TPR
      optimalFPR = FPR

    # print(f"Threshold: ---------------------- {threshold}")
    # print(f"True Positive Rate (Sensitivity): {TPR}")
    # print(f"False Positive Rate:              {FPR}")

  return optimalTPR, optimalFPR, optimalThreshold

globalOptimalModel = None
globalOptimalTPR = None
globalOptimalFPR = None
globalOptimalThreshold = None

for model_ in trainedModels:
    optimalTPR, optimalFPR, optimalThreshold = findOptimalThresholdForModel(
        model_,
        X_test_tensor,
        y_test_tensor
```

```
    )

    if (globalOptimalTPR == None):
      globalOptimalTPR = optimalTPR
      globalOptimalFPR = optimalFPR
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalTPR > globalOptimalTPR):
      globalOptimalTPR = optimalTPR
      globalOptimalFPR = optimalFPR
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalTPR == globalOptimalTPR and optimalFPR < globalOptimalFPR):
      globalOptimalFPR = optimalFPR
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold

print(globalOptimalModel)

gModel = globalOptimalModel['model']
with torch.no_grad(): output = gModel(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)

plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

gLr = globalOptimalModel['hyperparameters']['lr']
gEpochs = globalOptimalModel['hyperparameters']['epochs']
gbatchSize = globalOptimalModel['hyperparameters']['batchSize']
gLambda = globalOptimalModel['hyperparameters']['lambda']

with torch.no_grad(): output = gModel(X_test_tensor)
testPredictions = (np.array(output.detach().numpy().flatten()) >
globalOptimalThreshold).astype(int)
testError = np.mean(testPredictions != np.array(y_test).astype(int))

print(f'Global Optimal Model Hyperparameters: lr={gLr}, epochs={gEpochs}, batchSize=
{gbatchSize}, lamda={gLambda}')
print(f'Global Optimal TPR: {globalOptimalTPR}, Global Optimal FPR: {globalOptimalFPR}, Global
Optimal Threshold: {globalOptimalThreshold}')
print(f'Test Accuracy: {1-testError}, Test Error: {testError}')

"""### Re-produceability Test & Visualization"""

lr = globalOptimalModel['hyperparameters']['lr']
epochs = globalOptimalModel['hyperparameters']['epochs']
batchSize = globalOptimalModel['hyperparameters']['batchSize']
lambda_ = globalOptimalModel['hyperparameters']['lambda']


num_features = len(X_train_df.columns)
model = LogisticRegression(num_features, l2=lambda_)
lossFn = nn.BCELoss()

trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModelKFold(
    model,
    epochs,
    lossFn,
```

```
    X_data,
    y_data,
    lr=lr,
    batchSize=batchSize,
    k=3
)

model.eval()

plt.plot(trainAcc, label='train accuracy')
plt.plot(testAcc, label='test accuracy')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainErr, label='train error')
plt.plot(testErr, label='test error')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainLoss, label='train loss')
plt.plot(testLoss, label='test loss')
plt.grid(True)
plt.legend()

plt.show()


print(f'Train Accuracy: {trainAcc[-1]}, Train Error: {trainErr[-1]}, Train Loss: {trainLoss[-
1]}')
print(f'Test Accuracy: {testAcc[-1]}, Test Error: {testErr[-1]}, Test Loss: {testLoss[-1]}')

with torch.no_grad(): output = model(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)

plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

optimalTPR, optimalFPR, optimalThreshold = findOptimalThresholdForModel(
        { 'model': model },
        X_test_tensor,
        y_test_tensor
)

testPredictions = (np.array(output.detach().numpy().flatten()) > optimalThreshold).astype(int)
testError = np.mean(testPredictions != np.array(y_test).astype(int))

print(f'Re: Global Optimal Model Hyperparameters: lr={lr}, epochs={epochs}, batchSize=
{batchSize}, lambda={lambda_}')
print(f'Re: Global Optimal TPR: {optimalTPR}, Global Optimal FPR: {optimalFPR}, Global Optimal
Threshold: {optimalThreshold}')
print(f'Re: Test Accuracy: {1-testError}, Test Error: {testError}')

"""## Time Complexity Analysis"""

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
lambdaValue = 0.001
filteredModels = [modelInfo for modelInfo in trainedModels if modelInfo['hyperparameters']
['lambda'] == lambdaValue]

epochsList = [modelInfo['hyperparameters']['epochs'] for modelInfo in filteredModels]
learningRates = [modelInfo['hyperparameters']['lr'] for modelInfo in filteredModels]
batchSizes = [modelInfo['hyperparameters']['batchSize'] for modelInfo in filteredModels]
trainingTimes = [modelInfo['trainingTime'] for modelInfo in filteredModels]

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
sc = ax.scatter(epochsList, learningRates, batchSizes, c=trainingTimes, cmap='viridis',
marker='o', s=100)
ax.set_xlabel('Epochs')
ax.set_ylabel('Learning Rate')
ax.set_zlabel('Batch Size')
ax.set_title('Training Time Variation with Hyperparameters')
cbar = plt.colorbar(sc)
cbar.set_label('Training Time (seconds)')

plt.show()
```

```
# -*- coding: utf-8 -*-
"""SSHAH_TermProject_BreastCancerWisconsin_LogisticRegression_SGD_Final.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1kGumFmXu1XbalYaVxrBdY16Kn5ZlgHaL

## Importing Data and Libraries
"""

!pip install ucimlrepo

from ucimlrepo import fetch_ucirepo
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
from sklearn.decomposition import PCA
from sklearn.model_selection import KFold

import matplotlib.pyplot as plt
import seaborn as sns
import torch
import torch.nn as nn

import copy
import time

torch.manual_seed(42)
np.random.seed(42)

breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)

data = breast_cancer_wisconsin_diagnostic.data.features
data['labels'] = breast_cancer_wisconsin_diagnostic.data.targets

print(data.info())

"""## Pre-processing Data: Cleaning, Equalizing, and LabelEncoding"""

missing = data.isnull().sum()
duplicates = data.duplicated().sum()

data = data.dropna()
data = data.drop_duplicates()

print("Missing values removed:\n" + str(missing))
print("\nDuplicates removed:", duplicates)

# Balancing the data here to remove the class imbalance. Tried exploring class_weights to
address imbalance, but ran into issues with Pytorch.
# Hence used this approach as an alternative

classCount = data['labels'].value_counts()
minClassCount = classCount.min()
newData = pd.DataFrame(columns=data.columns)

for label in classCount.index:
      classSamples = data[data['labels'] == label]
      newSamples = classSamples.sample(minClassCount)
      newData = pd.concat([newData, newSamples])

data = newData
print(data.info())
```

```
# Encoding the labels with 0 and 1

featuresRaw = data.drop('labels', axis=1)
labelsRaw = data.labels

labelMapping = {}

labelEncoder = LabelEncoder()
labelsEncoded = labelEncoder.fit_transform(labelsRaw)

for class_ in labelEncoder.classes_:
  encoding = labelEncoder.transform([class_])[0]
  labelMapping[encoding] = class_

labels = labelsEncoded
print(labelMapping)

"""## Splitting Data"""

X_train, X_test, y_train, y_test = train_test_split(featuresRaw, labels, test_size=0.3,
random_state=42)

trainScaler = StandardScaler()
X_train = trainScaler.fit_transform(X_train)
X_test = trainScaler.transform(X_test)

X_train_df = pd.DataFrame(X_train, columns=featuresRaw.columns)
X_test_df = pd.DataFrame(X_test, columns=featuresRaw.columns)

X_train_df.head()

X_data = np.array(featuresRaw)
y_data = np.array(labels)

"""## Feature Visualization"""

# Visualizing the spread of data-points of feature for the two classes. Interesting to observe
how they take a guassian shape, and
# gives an intutive sense of how a high-dimensional decision boundarys might separate them.

featuresLabelsDf = copy.deepcopy(X_train_df)
featuresLabelsDf['labels'] = copy.deepcopy(y_train)

col = len(featuresLabelsDf.columns) - 1
num_rows = (col - 1) // 4 + 1
num_cols = min(col, 4)

fig, axes = plt.subplots(nrows=num_rows, ncols=num_cols, figsize=(24, 22))
for i, column in enumerate(featuresLabelsDf.columns[:-1]):
    row = i // num_cols
    col = i % num_cols
    sns.histplot(featuresLabelsDf, x=column, hue='labels', kde=True, multiple='stack',
bins=30, ax=axes[row, col])
    axes[row,col].set_title('Distribution of ' + str(column) + ' by Class')
    axes[row,col].set_xlabel(column)
    axes[row,col].set_ylabel('Frequency')
    axes[row,col].legend(title='Class', labels=['Benign', 'Malignant'])

plt.tight_layout()
plt.show()

"""## Model Construction and Training"""

class LogisticRegression(nn.Module):
  def __init__(self, num_features, l2=0.0):
    super(LogisticRegression, self).__init__()
```

```
    self.layer1 = nn.Linear(num_features, 1)
    self.sig = nn.Sigmoid()
    self.l2 = l2


  def forward(self, x):
    out0 = self.layer1(x)
    out1 = self.sig(out0)
    return out1

# Helper Functions that perform training and reseting of model weights between iteration

def resetWeights(model):
    for layer in model.children():
        if isinstance(layer, nn.Linear):
            layer.reset_parameters()

def trainModelKFold(model, epochs, lossFn, optimizer, X, y, decisionThreshold=0.5, k=1):
    trainAcc, testAcc = np.zeros(epochs), np.zeros(epochs)
    trainErr, testErr = np.zeros(epochs), np.zeros(epochs)
    trainLoss, testLoss = np.zeros(epochs), np.zeros(epochs)

    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        trainScaler = StandardScaler()
        X_train = trainScaler.fit_transform(X_train)
        X_test = trainScaler.transform(X_test)

        X_train_tensor = torch.tensor(X_train).float()
        X_test_tensor = torch.tensor(X_test).float()
        y_train_tensor = torch.tensor(y_train).float()
        y_test_tensor = torch.tensor(y_test).float()

        resetWeights(model)
        for epoch in range(epochs):
            output = model(X_train_tensor)
            loss = lossFn(output.squeeze(), y_train_tensor)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            with torch.no_grad():
                trainLoss[epoch] += loss.item()
                trainPredictions = (np.array(output.flatten()) >
decisionThreshold).astype(int)
                trainError = np.mean(trainPredictions != np.array(y_train_tensor).astype(int))


                output = model(X_test_tensor)
                loss = lossFn(output.squeeze(), y_test_tensor)

                testLoss[epoch] += loss.item()
                testPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
                testError = np.mean(testPredictions != np.array(y_test_tensor).astype(int))

                trainErr[epoch] += trainError.item()
                testErr[epoch] += testError.item()
                trainAcc[epoch] += 1 - trainError.item()
                testAcc[epoch] += 1 - testError.item()

    avg_trainAcc = trainAcc/3
    avg_testAcc = testAcc/3
    avg_trainErr = trainErr/3
```

```
    avg_testErr = testErr/3
    avg_trainLoss = trainLoss/3
    avg_testLoss = testLoss/3

    return avg_trainAcc, avg_testAcc, avg_trainErr, avg_testErr, avg_trainLoss, avg_testLoss

def trainModel(model, epochs, lossFn, optimizer, X_train, X_test, y_train, y_test,
decisionThreshold=0.5):
    trainAcc, testAcc = [], []
    trainErr, testErr = [], []
    trainLoss, testLoss = [], []

    resetWeights(model)
    for epoch in range(epochs):
        output = model(X_train)
        loss = lossFn(output.squeeze(), y_train)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        with torch.no_grad():
            trainLoss.append(loss.item())
            trainPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
            trainError = np.mean(trainPredictions != np.array(y_train).astype(int))


            output = model(X_test)
            loss = lossFn(output.squeeze(), y_test)

            testLoss.append(loss.item())
            testPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
            testError = np.mean(testPredictions != np.array(y_test).astype(int))

            trainErr.append(trainError.item())
            testErr.append(testError.item())
            trainAcc.append(1 - trainError.item())
            testAcc.append(1 - testError.item())

    return trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss


X_train_tensor = torch.tensor(X_train).float()
y_train_tensor = torch.tensor(y_train).float()
X_test_tensor = torch.tensor(X_test).float()
y_test_tensor = torch.tensor(y_test).float()

"""## Grid Search Analysis"""

# Grid search for find the hyperparameter tuning to find best model

epochs_list = [50, 100, 500, 1000]
learning_rates = [1, 0.1, 0.01, 0.001]
momentums = [0, 0.5, 0.9, 0.99]
lambdas = [0, 0.001, 0.01, 0.1, 1]
trainedModels = []

num_features = len(X_train_df.columns)
lossFn = nn.BCELoss()

for lambda_ in lambdas:
  model = LogisticRegression(num_features, l2=lambda_)
  for momentum in momentums:
    for lr in learning_rates:
        optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum,
weight_decay=model.l2)
        for epochs in epochs_list:
          startTime = time.time()
```

```
          trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModelKFold(
              model,
              epochs,
              lossFn,
              optimizer,
              X_data,
              y_data,
              k=3,
          )
          trainingTime = time.time() - startTime
          trainedModels.append({ 'model': copy.deepcopy(model),
                                 'hyperparameters': {'epochs': epochs, 'lr': lr, 'momentum':
momentum, 'lambda': lambda_ },
                                 'trainingStats': { 'testAcc': testAcc[-1], 'trainAcc':
trainAcc[-1],
                                                    'testError': testErr[-1], 'trainError':
trainErr[-1],
                                                    'testLoss': testLoss[-1], 'trainLoss':
trainLoss[-1] },
                                 'trainingTime': trainingTime
                                 })
          print(f'Completed epochs, lr, momentum, lambda, test error: {epochs}, {lr},
{momentum}, {lambda_}, --- {testAcc[-1]}')

"""## Finding the best model that maximizes accuracy using ROC curve:"""

def findOptimalThresholdForModelAcc(model, X_test_tensor, y_test_tensor):
  with torch.no_grad(): output = model(X_test_tensor)
  _, _, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
  optimalAcc = None
  optimalThreshold = None

  for threshold in thresholds:
    testPredictions = (np.array(output.detach().numpy().flatten()) > threshold).astype(int)
    testAcc = 1 - np.mean(testPredictions != np.array(y_test_tensor).astype(int))
    if (optimalThreshold == None):
      optimalThreshold = threshold
      optimalAcc = testAcc
    elif (testAcc > optimalAcc):
      optimalThreshold = threshold
      optimalAcc = testAcc



  return optimalAcc, optimalThreshold

globalOptimalModel = None
globalOptimalLoss = None
globalOptimalAcc = None
globalOptimalThreshold = None

for model_ in trainedModels:
    optimalAcc, optimalThreshold = findOptimalThresholdForModelAcc(
        model_['model'],
        X_test_tensor,
        y_test_tensor
    )
    optimalLoss = model_['trainingStats']['testLoss']

    if (globalOptimalLoss == None):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalAcc > globalOptimalAcc):
      globalOptimalLoss = optimalLoss
```

```
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalAcc == globalOptimalAcc and optimalLoss < globalOptimalLoss):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold

print(globalOptimalModel)

gModel = globalOptimalModel['model']
with torch.no_grad(): output = gModel(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)

plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

gLr = globalOptimalModel['hyperparameters']['lr']
gEpochs = globalOptimalModel['hyperparameters']['epochs']
gMomentum = globalOptimalModel['hyperparameters']['momentum']
gLambda = globalOptimalModel['hyperparameters']['lambda']

index = int(np.where(thresholds == globalOptimalThreshold)[0])
FPR = fpr[index]
TPR = tpr[index]

print(f'Global Optimal Model Hyperparameters: lr={gLr}, epochs={gEpochs}, momentum={gMomentum},
lamda={gLambda}')
print(f'Global Optimal TPR: {TPR}, Global Optimal FPR: {FPR}, Global Optimal Threshold:
{globalOptimalThreshold}')
print(f'Global Optimal Acc: {globalOptimalAcc}, Global Optimal Loss: {globalOptimalLoss}')

"""### Reproduceability Test & Visualizing Curves"""

lr = globalOptimalModel['hyperparameters']['lr']
epochs = globalOptimalModel['hyperparameters']['epochs']
momentum = globalOptimalModel['hyperparameters']['momentum']
lambda_ = globalOptimalModel['hyperparameters']['lambda']

num_features = len(X_train_df.columns)
model = LogisticRegression(num_features, l2=lambda_)
lossFn = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum,
weight_decay=model.l2)

trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModelKFold(
    model,
    epochs,
    lossFn,
    optimizer,
    X_data,
    y_data,
    k=3,
)

model.eval()
```

```
plt.plot(trainAcc, label='train accuracy')
plt.plot(testAcc, label='test accuracy')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainErr, label='train error')
plt.plot(testErr, label='test error')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainLoss, label='train loss')
plt.plot(testLoss, label='test loss')
plt.grid(True)
plt.legend()

plt.show()


print(f'Train Accuracy: {trainAcc[-1]}, Train Error: {trainErr[-1]}, Train Loss: {trainLoss[-
1]}')
print(f'Test Accuracy: {testAcc[-1]}, Test Error: {testErr[-1]}, Test Loss: {testLoss[-1]}')

with torch.no_grad(): output = model(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)

plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

optimalAcc, optimalThreshold = findOptimalThresholdForModelAcc(
        model,
        X_test_tensor,
        y_test_tensor,
    )

index = int(np.where(thresholds == optimalThreshold)[0])
FPR = fpr[index]
TPR = tpr[index]

print(f'Global Optimal Model Hyperparameters: lr={lr}, epochs={epochs}, momentum={momentum},
lamda={lambda_}')
print(f'Global Optimal TPR: {TPR}, Global Optimal FPR: {FPR}, Global Optimal Threshold:
{optimalThreshold}')
print(f'Global Optimal Acc: {optimalAcc}, Global Optimal Loss: {testLoss[-1]}')

"""## Risk Minimization: Finding the Optimal Model such that it minimizes the risk by
increasing true positive rate (TPR)"""

def findOptimalThresholdForModel(model, X_test_tensor, y_test_tensor, fprThreshhold=0.25):
  with torch.no_grad(): output = model['model'](X_test_tensor).detach().numpy().flatten()
  fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(), output, pos_label=1)
  optimalTPR = None
  optimalFPR = None
  optimalThreshold = None

  for threshold in thresholds:
    index = int(np.where(thresholds == threshold)[0])
```

```
    FPR = fpr[index]
    TPR = tpr[index]

    if (optimalTPR == None):
      optimalThreshold = threshold
      optimalTPR = TPR
      optimalFPR = FPR
    elif (TPR > optimalTPR and FPR < fprThreshhold):
      optimalThreshold = threshold
      optimalTPR = TPR
      optimalFPR = FPR

    # print(f"Threshold: ---------------------- {threshold}")
    # print(f"True Positive Rate (Sensitivity): {TPR}")
    # print(f"False Positive Rate:              {FPR}")

  return optimalTPR, optimalFPR, optimalThreshold

globalOptimalModel = None
globalOptimalTPR = None
globalOptimalFPR = None
globalOptimalThreshold = None

for model_ in trainedModels:
    optimalTPR, optimalFPR, optimalThreshold = findOptimalThresholdForModel(
        model_,
        X_test_tensor,
        y_test_tensor
    )

    if (globalOptimalTPR == None):
      globalOptimalTPR = optimalTPR
      globalOptimalFPR = optimalFPR
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalTPR > globalOptimalTPR):
      globalOptimalTPR = optimalTPR
      globalOptimalFPR = optimalFPR
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalTPR == globalOptimalTPR and optimalFPR < globalOptimalFPR):
      globalOptimalFPR = optimalFPR
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold

print(globalOptimalModel)

gModel = globalOptimalModel['model']
with torch.no_grad(): output = gModel(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)

plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

gLr = globalOptimalModel['hyperparameters']['lr']
gEpochs = globalOptimalModel['hyperparameters']['epochs']
gMomentum = globalOptimalModel['hyperparameters']['momentum']
gLambda = globalOptimalModel['hyperparameters']['lambda']
```

```
with torch.no_grad(): output = gModel(X_test_tensor)
testPredictions = (np.array(output.detach().numpy().flatten()) >
globalOptimalThreshold).astype(int)
testError = np.mean(testPredictions != np.array(y_test).astype(int))

print(f'Global Optimal Model Hyperparameters: lr={gLr}, epochs={gEpochs}, momentum={gMomentum},
lamda={gLambda}')
print(f'Global Optimal TPR: {globalOptimalTPR}, Global Optimal FPR: {globalOptimalFPR}, Global
Optimal Threshold: {globalOptimalThreshold}')
print(f'Test Accuracy: {1-testError}, Test Error: {testError}')

"""### Re-produceability Test & Visualization"""

lr = globalOptimalModel['hyperparameters']['lr']
epochs = globalOptimalModel['hyperparameters']['epochs']
momentum = globalOptimalModel['hyperparameters']['momentum']
lambda_ = globalOptimalModel['hyperparameters']['lambda']


num_features = len(X_train_df.columns)
model = LogisticRegression(num_features, l2=lambda_)
lossFn = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum,
weight_decay=model.l2)

trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModelKFold(
    model,
    epochs,
    lossFn,
    optimizer,
    X_data,
    y_data,
    k=3
)

model.eval()

plt.plot(trainAcc, label='train accuracy')
plt.plot(testAcc, label='test accuracy')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainErr, label='train error')
plt.plot(testErr, label='test error')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainLoss, label='train loss')
plt.plot(testLoss, label='test loss')
plt.grid(True)
plt.legend()

plt.show()


print(f'Train Accuracy: {trainAcc[-1]}, Train Error: {trainErr[-1]}, Train Loss: {trainLoss[-
1]}')
print(f'Test Accuracy: {testAcc[-1]}, Test Error: {testErr[-1]}, Test Loss: {testLoss[-1]}')

with torch.no_grad(): output = model(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)
```

```
plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

optimalTPR, optimalFPR, optimalThreshold = findOptimalThresholdForModel(
        { 'model': model },
        X_test_tensor,
        y_test_tensor
)

testPredictions = (np.array(output.detach().numpy().flatten()) > optimalThreshold).astype(int)
testError = np.mean(testPredictions != np.array(y_test).astype(int))

print(f'Re: Global Optimal Model Hyperparameters: lr={lr}, epochs={epochs}, momentum=
{momentum}, lambda={lambda_}')
print(f'Re: Global Optimal TPR: {optimalTPR}, Global Optimal FPR: {optimalFPR}, Global Optimal
Threshold: {optimalThreshold}')
print(f'Re: Test Accuracy: {1-testError}, Test Error: {testError}')

"""## Time Complexity Analysis"""

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

lambdaValue = 0.001
filteredModels = [modelInfo for modelInfo in trainedModels if modelInfo['hyperparameters']
['lambda'] == lambdaValue]

epochsList = [modelInfo['hyperparameters']['epochs'] for modelInfo in filteredModels]
learningRates = [modelInfo['hyperparameters']['lr'] for modelInfo in filteredModels]
momentums = [modelInfo['hyperparameters']['momentum'] for modelInfo in filteredModels]
trainingTimes = [modelInfo['trainingTime'] for modelInfo in filteredModels]

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
sc = ax.scatter(epochsList, learningRates, momentums, c=trainingTimes, cmap='viridis',
marker='o', s=100)
ax.set_xlabel('Epochs')
ax.set_ylabel('Learning Rate')
ax.set_zlabel('Momentum')
ax.set_title('Training Time Variation with Hyperparameters')
cbar = plt.colorbar(sc)
cbar.set_label('Training Time (seconds)')

plt.show()
```

2) Support Vector Machine: Steps to run the code:

1) Run the file `Wisconsin_SVM.ipynb` in the directory Wisconsin/SVM. For running the file, the user just have to execute the "Run All Below" command, mentioned in the dropdown menu of the Cell tab in the Jupyter notebook.

```
# -*- coding: utf-8 -*-
"""Wisconsin_SVM.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/15SnRzMoCX-36bfBSmDRcynq5Hyej81jo
"""

!pip3 install ucimlrepo
from ucimlrepo import fetch_ucirepo
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import validation_curve
from sklearn.model_selection import learning_curve
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd

breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)

X = breast_cancer_wisconsin_diagnostic.data.features
y = breast_cancer_wisconsin_diagnostic.data.targets

print(breast_cancer_wisconsin_diagnostic.metadata)

print(breast_cancer_wisconsin_diagnostic.variables)

# Loading the data from the CSV file
data_url = 'https://archive.ics.uci.edu/static/public/17/data.csv'
df = pd.read_csv(data_url, index_col='ID')

print(df.head())

X = df.drop('Diagnosis', axis=1)
y = df['Diagnosis']

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardizing the features using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("\nScaled Training Data:")
print(pd.DataFrame(X_train_scaled, columns=X.columns).head())

# Performing SVM without kernel

# Defining the hyperparameter
param_grid_linear = {
    'C': list(range(1, 11))
}

svm_linear = SVC(kernel='linear')

grid_search_linear = GridSearchCV(estimator=svm_linear, param_grid=param_grid_linear,
                                  cv=5, scoring='accuracy', verbose=2, n_jobs=-1)
grid_search_linear.fit(X_train_scaled, y_train)

best_svm_linear = grid_search_linear.best_estimator_
```

```
y_pred_linear = best_svm_linear.predict(X_test_scaled)

# Evaluating the performance of the linear SVM
accuracy_linear = accuracy_score(y_test, y_pred_linear)
print("Best Linear SVM Accuracy:", accuracy_linear)
print("Classification Report (Best Linear SVM):")
print(classification_report(y_test, y_pred_linear))
best_C_linear = grid_search_linear.best_params_['C']
print("Best C:", best_C_linear)

C_values = list(range(1, 11))
train_scores, test_scores = [], []

for C in C_values:
    best_svm_linear.C = C
    best_svm_linear.fit(X_train_scaled, y_train)

    train_score = best_svm_linear.score(X_train_scaled, y_train)
    train_scores.append(train_score)

    test_score = best_svm_linear.score(X_test_scaled, y_test)
    test_scores.append(test_score)

# Plotting the training and test scores against the hyperparameter C
plt.figure(figsize=(10, 6))
plt.plot(C_values, train_scores, 'o-', label='Training Score')
plt.plot(C_values, test_scores, 'o-', label='Test Score')
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Test Scores vs C')
plt.show()


# Plotting the learning curve
def plot_learning_curve(estimator, X, y, cv, train_sizes=np.linspace(0.1, 1.0, 10)):
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, train_sizes=train_sizes, scoring='accuracy', n_jobs=-1
    )

    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_scores_mean, 'o-', color='r', label='Training Score')
    plt.plot(train_sizes, test_scores_mean, 'o-', color='g', label='Validation Score')

    plt.title('Learning Curve')
    plt.xlabel('Training Examples')
    plt.ylabel('Score')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()

plot_learning_curve(best_svm_linear, X_train_scaled, y_train, cv=5)

# Performing the Polynomial SVM

# Defining the hyperparameters
param_grid = {
    'C': list(range(1, 11)),
    'degree': list(range(1, 31)),
}
svm_poly = SVC(kernel='poly')
```

```
grid_search = GridSearchCV(estimator=svm_poly, param_grid=param_grid, cv=5,
scoring='accuracy', verbose=2, n_jobs=-1)

grid_search.fit(X_train_scaled, y_train)

best_svm_poly = grid_search.best_estimator_

y_pred_poly = best_svm_poly.predict(X_test_scaled)

# Evaluating the performance of Polynomial Kernel SVM
accuracy_poly = accuracy_score(y_test, y_pred_poly)
print("Best Polynomial Kernel SVM Accuracy:", accuracy_poly)
print("Classification Report (Best Polynomial Kernel SVM):")
print(classification_report(y_test, y_pred_poly))
best_degree = grid_search.best_params_['degree']
print("Best Degree:", best_degree)
best_C = grid_search.best_params_['C']
print("Best C:", best_C)

degrees = list(range(1, 31))
train_scores, test_scores = [], []

for degree in degrees:
    best_svm_poly.degree = degree
    best_svm_poly.fit(X_train_scaled, y_train)

    train_score = best_svm_poly.score(X_train_scaled, y_train)
    train_scores.append(train_score)

    test_score = best_svm_poly.score(X_test_scaled, y_test)
    test_scores.append(test_score)

# Plotting the training and test scores against polynomial degree
plt.figure(figsize=(10, 6))
plt.plot(degrees, train_scores, 'o-', label='Training Score')
plt.plot(degrees, test_scores, 'o-', label='Test Score')
plt.xlabel('Degree of Polynomial')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Test Scores vs Polynomial Degree for SVM')
plt.show()

# Plotting the learning curve
def plot_learning_curve(estimator, X, y, cv, train_sizes=np.linspace(0.1, 1.0, 10)):
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, train_sizes=train_sizes, scoring='accuracy', n_jobs=-1
    )

    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_scores_mean, 'o-', color='r', label='Training Score')
    plt.plot(train_sizes, test_scores_mean, 'o-', color='g', label='Validation Score')

    plt.title('Learning Curve')
    plt.xlabel('Training Examples')
    plt.ylabel('Score')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()

plot_learning_curve(best_svm_poly, X_train_scaled, y_train, cv=5)
```

```
# Performing SVM with RBF Kernel

# Defining the hyperparameters
param_grid_rbf = {
    'C': list(range(1, 11)),
    'gamma': np.arange(0.1, 2.1, 0.1)
}

svm_rbf = SVC(kernel='rbf')

grid_search_rbf = GridSearchCV(estimator=svm_rbf, param_grid=param_grid_rbf, cv=5,
scoring='accuracy', verbose=2, n_jobs=-1)
grid_search_rbf.fit(X_train_scaled, y_train)

best_svm_rbf = grid_search_rbf.best_estimator_

y_pred_rbf = best_svm_rbf.predict(X_test_scaled)

# Evaluating the performance of RBF SVM
accuracy_rbf = accuracy_score(y_test, y_pred_rbf)
print("Best RBF SVM Accuracy:", accuracy_rbf)
print("Classification Report (Best RBF SVM):")
print(classification_report(y_test, y_pred_rbf))
best_gamma = grid_search_rbf.best_params_['gamma']
print("Best Gamma:", best_gamma)
best_C = grid_search.best_params_['C']
print("Best C:", best_C)

gammas = np.arange(0.1, 2.1, 0.1)
train_scores, test_scores = [], []

for gamma in gammas:
    best_svm_rbf.gamma = gamma
    best_svm_rbf.fit(X_train_scaled, y_train)

    train_score = best_svm_rbf.score(X_train_scaled, y_train)
    train_scores.append(train_score)

    test_score = best_svm_rbf.score(X_test_scaled, y_test)
    test_scores.append(test_score)

# Plotting the training and test scores against RBF SVM gamma
plt.figure(figsize=(10, 6))
plt.plot(gammas, train_scores, 'o-', label='Training Score')
plt.plot(gammas, test_scores, 'o-', label='Test Score')
plt.xlabel('Gamma of RBF')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Test Scores vs Gamma for RBF SVM')
plt.show()


# Plotting the learning curve for RBF SVM
def plot_learning_curve_rbf(estimator, X, y, cv, train_sizes=np.linspace(0.1, 1.0, 10)):
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, train_sizes=train_sizes, scoring='accuracy', n_jobs=-1
    )

    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_scores_mean, 'o-', color='r', label='Training Score')
    plt.plot(train_sizes, test_scores_mean, 'o-', color='g', label='Validation Score')
```

```
    plt.title('Learning Curve (RBF SVM)')
    plt.xlabel('Training Examples')
    plt.ylabel('Score')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()

plot_learning_curve_rbf(best_svm_rbf, X_train_scaled, y_train, cv=5)

# Computing the confusion matrix for the selected SVM model

conf_matrix = confusion_matrix(y_test, y_pred_linear)

# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['B', 'M'],
yticklabels=['B', 'M'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

3) k-Nearest Neighbours: Steps to run the code:

1) Run the file `TMahmood_TermProject_BreastCancerWisconsin_kNN.ipynb` in the directory Wisconsin/kNN. For running the file, the user just have to execute the "Run All Below" command, mentioned in the dropdown menu of the Cell tab in the Jupyter notebook.

```
#!/usr/bin/env python
# coding: utf-8

# ## 1. Importing Libraries

# In[19]:


# !pip install ucimlrepo

from ucimlrepo import fetch_ucirepo
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import validation_curve
from sklearn.model_selection import learning_curve
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
import seaborn as sns
import numpy as np
import pandas as pd


# ## 2. Importing Data

# In[2]:


breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)

X = breast_cancer_wisconsin_diagnostic.data.features
y = breast_cancer_wisconsin_diagnostic.data.targets

merged_df = pd.merge(X, y, left_index=True, right_index=True)

merged_df.head()


# ## 3. Data Visualization

# Generating histograms with kernel density estimates for numerical columns in the entire dataset. There are clear distinctions in the distribution of Malignant and Benign for most fea

# In[3]:


numerical_columns = merged_df.select_dtypes(include=['float64']).columns

num_rows = len(numerical_columns) // 4 + 1
num_cols = min(len(numerical_columns), 4)

plt.figure(figsize=(15, 5 * num_rows))

for i, column in enumerate(numerical_columns, start=1):
    plt.subplot(num_rows, num_cols, i)
    sns.histplot(merged_df, x=column, hue='Diagnosis', kde=True, element='step', common_norm=False, palette='muted')
    plt.title(column)

plt.tight_layout()
plt.show()


# ## 4. Data Preprocessing

# ### 4.1. Checking Null Values

# Checking the dataset for the presence of null values confirms that there are none in this particular dataset.

# In[4]:


merged_df.replace("?", np.nan, inplace = True)

null_values = merged_df.isnull().sum()

print(null_values)


# ### 4.2. Checking Data set Type

# Examining the data types of each feature in the dataset to ensure the absence of data types that could potentially impact performance adversely.

# In[5]:


print("Data Types:")
print(X.dtypes)

print("Target Data Types:")
print(y.dtypes)


# ### 4.3. Scalarization of the Dataset

# Applying standardization to the feature matrix (X) using StandardScaler, creating X_scaled_df as a DataFrame. This ensures consistent scaling, aiding machine learning model performan

# In[6]:


scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)
X_scaled_df.head()


# ### 4.4. Label Encoding

# Transforming the target variable 'y' by replacing 'M' with 1 and 'B' with 0 for compatibility with binary classification models.

# In[7]:
```

```
y = y.replace({'M': 1, 'B': 0})


# ### 4.4. Principal Component Analysis

# Using Principal Component Analysis (PCA) on the scaled features (X_scaled_df), visualizing cumulative explained variance of principal components for dimensionality reduction.

# In[8]:


pca = PCA()
X_pca = pca.fit_transform(X_scaled_df)

explained_variance_ratio = pca.explained_variance_ratio_

cumulative_variance = np.cumsum(explained_variance_ratio)

plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, marker='o', linestyle='--')
plt.title('Cumulative Explained Variance')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Variance Explained')
plt.grid(True)
plt.show()


# Visualizing the explained variance ratio for each principal component using a bar plot and cumulative step plot.

# In[9]:


plt.bar(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio, alpha=0.75, align='center')
plt.step(range(1, len(explained_variance_ratio) + 1), np.cumsum(explained_variance_ratio), where='mid')
plt.title('Explained Variance Ratio for Each Principal Component')
plt.xlabel('Principal Component Number')
plt.ylabel('Explained Variance Ratio')
plt.show()


# Creating a bar plot to visualize the absolute loadings of Principal Component 1 (PC1) in a PCA-transformed dataset, highlighting key features and their contributions to the variance

# In[10]:


feature_names = X.columns.tolist()

loadings_df = pd.DataFrame(pca.components_, columns=feature_names)

loadings_df = loadings_df[feature_names]

sorted_feature_names = loadings_df.abs().iloc[0, :].sort_values(ascending=False).index
sorted_loadings = loadings_df[sorted_feature_names]

plt.bar(sorted_feature_names, np.abs(sorted_loadings.iloc[0, :]))
plt.title('Absolute Loadings for PC')
plt.xlabel('Features')
plt.ylabel('Absolute Loading')
plt.xticks(rotation=45, ha='right')
plt.show()


# Performing PCA to retain 90% of the variance in the scaled dataset.

# In[11]:


cumulative_variance_ratio = np.cumsum(pca.explained_variance_ratio_)
n_components_90 = np.argmax(cumulative_variance_ratio >= 0.9) + 1

pca = PCA(n_components=n_components_90)
X_pca = pca.fit_transform(X_scaled_df)

columns_pca = [f"PC_{i+1}" for i in range(X_pca.shape[1])]
pca_df = pd.DataFrame(data=X_pca, columns=columns_pca)
pca_df.head()
y=y["Diagnosis"]


# ## 5. Model Training (Grid Search)

# ### 5.1. Defining the model and hyperparameters

# Defining the function to initiate kFold and knn, and assiging the hyperparameter range in the param_grid dictionary.

# In[32]:


kf = KFold(n_splits=10, shuffle=True, random_state=22)
knn = KNeighborsClassifier()
X_train, X_test, y_train, y_test = train_test_split(pca_df, y, test_size=0.2, random_state=42)
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29],
    'weights': ['uniform', 'distance'],
    'p': [i/2 for i in range(2, 9)],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'leaf_size': [10, 20, 30, 40],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}


# In[15]:


grid_search = GridSearchCV(knn, param_grid, cv=kf, scoring='accuracy')

results = grid_search.fit(X_train, y_train)


# ### 5.2. Determing the best model based on validation accuracy

# Examining different hyperparameter combinations and their accuracies through a grid search. The bar plot displays the top 5 configurations with their accuracy scores, helping identif

# In[45]:


hyperparameters = []
accuracies = []

for params, mean_score, _ in zip(grid_search.cv_results_['params'], grid_search.cv_results_['mean_test_score'], grid_search.cv_results_['std_test_score']):
```

```
    params_tuple = tuple(params.values())

    hyperparameters.append(params_tuple)
    accuracies.append(mean_score)

results_df = pd.DataFrame({'Hyperparameters': hyperparameters, 'Accuracy': accuracies})

results_df = results_df.sort_values(by='Accuracy', ascending=False)

top_n = 5
ax = sns.barplot(x='Accuracy', y='Hyperparameters', data=results_df.head(top_n), orient='h')

for index, value in enumerate(results_df.head(top_n)['Accuracy']):
    ax.text(value, index, f'{value:.3f}', ha='left', va='center', color='black')

ax.set_xlim(0, 1.1 * max(results_df['Accuracy']))  # Adjust the multiplier as needed

plt.title(f'Top {top_n} Accuracy Across Different Hyperparameter Values')
plt.xlabel('Accuracy')
plt.ylabel('Hyperparameters')
plt.xlim(0.6, 1)

plt.show()


# In[37]:


best_hyperparameters = results_df.iloc[0]['Hyperparameters']

print(f"Best Hyperparameters: "
      f"algorithm={best_hyperparameters[0]}, "
      f"leaf_size={best_hyperparameters[1]}, "
      f"metric={best_hyperparameters[2]}, "
      f"n_neighbors={best_hyperparameters[3]}, "
      f"p={best_hyperparameters[4]}, "
      f"weights={best_hyperparameters[5]}")


# ### 5.3. Examining the performance of the best model

# Using the best hyperparameters obtained from a grid search, metrics such as accuracy, precision, recall, and F1 score are computed and presented, along with a heatmap visualization o

# In[39]:


best_knn = KNeighborsClassifier(
    algorithm=best_hyperparameters[0],
    leaf_size=best_hyperparameters[1],
    metric=best_hyperparameters[2],
    n_neighbors=best_hyperparameters[3],
    p=best_hyperparameters[4],
    weights=best_hyperparameters[5]
)

best_knn.fit(X_train, y_train)

y_pred = best_knn.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

print(f'Best Model Accuracy on Test Set: {accuracy:.4f}')
print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1 Score: {f1:.4f}')

conf_matrix = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(4, 3))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()


# ## 6. Model Training (Iterative Search)

# ### 6.1 Calling the helper functions for model training

# Defining functions for KNN cross-validation, training on folds, and finding optimal hyperparameter values through averaging test accuracies.

# In[21]:


def knn_cross_val_fold(X_train_fold, y_train_fold, X_val_fold, y_val_fold, knn):
    knn.fit(X_train_fold, y_train_fold)

    y_train_fold_pred = knn.predict(X_train_fold)
    y_val_fold_pred = knn.predict(X_val_fold)

    fold_train_accuracy = accuracy_score(y_train_fold, y_train_fold_pred)
    fold_test_accuracy = accuracy_score(y_val_fold, y_val_fold_pred)

    return fold_train_accuracy, fold_test_accuracy

def knn_cross_val(X_train, y_train, knn, kf=kf):
    fold_train_accuracies = []
    fold_test_accuracies = []

    for train_index, test_index in kf.split(X_train):
        X_train_fold, X_val_fold = X_train.iloc[train_index], X_train.iloc[test_index]
        y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.iloc[test_index]

        fold_train_accuracy, fold_test_accuracy = knn_cross_val_fold(X_train_fold, y_train_fold, X_val_fold, y_val_fold, knn)

        fold_train_accuracies.append(fold_train_accuracy)
        fold_test_accuracies.append(fold_test_accuracy)

    avg_train_accuracy = np.mean(fold_train_accuracies)
    avg_test_accuracy = np.mean(fold_test_accuracies)

    return avg_train_accuracy, avg_test_accuracy

def find_best_value(test_accuracies, values_range):
    best_index = np.argmax(test_accuracies)
    best_value = values_range[best_index]
```

```
    return best_index, best_value


# ### 6.2. Nearest Neighbors Hyperparameter Tuning Curve

# Iterating KNN models with different neighbor values (3 to 29) using 10-fold cross-validation.

# In[22]:


knn = KNeighborsClassifier()

n_neighbors_values = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

train_accuracies = []
test_accuracies = []

for n_neighbors in n_neighbors_values:
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_index, best_n_neighbors = find_best_value(test_accuracies, n_neighbors_values)

plt.plot(n_neighbors_values, train_accuracies, label='Average Training Accuracy')
plt.plot(n_neighbors_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_n_neighbors, test_accuracies[best_index], color='red', marker='x', label=f'Best Test Accuracy (n_neighbors={best_n_neighbors})')
plt.xlabel('Number of Neighbors (n_neighbors)')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different n_neighbors Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best n_neighbors: {best_n_neighbors}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_index]}")


# ### 6.3. Weights Methods Hyperparameter Tuning Curve

# Iterating KNN models with different distance weights methods (uniform and distance) using 10-fold cross-validation.

# In[25]:


weights_values = ['uniform', 'distance']

train_accuracies = []
test_accuracies = []

for weights in weights_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=weights)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_weights_index, best_weights = find_best_value(test_accuracies, weights_values)

plt.plot(weights_values, train_accuracies, label='Average Training Accuracy')
plt.plot(weights_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_weights, test_accuracies[best_index], color='red', marker='x', label=f'Best Test Accuracy (weights={best_weights})')
plt.xlabel('Weights')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Weights Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best weights: {best_weights}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_weights_index]}")


# ### 6.4. Minkowski distance p values Hyperparameter Tuning Curve

# Iterating KNN models with different p values for minkowski distance (0.5 to 4.0) using 10-fold cross-validation.

# In[26]:


p_values = [i/2 for i in range(2, 9)]

train_accuracies = []
test_accuracies = []

for p in p_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=p, metric='minkowski')
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_p_index, best_p = find_best_value(test_accuracies, p_values)

plt.plot(p_values, train_accuracies, label='Average Training Accuracy')
plt.plot(p_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_p, test_accuracies[best_p_index], color='red', marker='x', label=f'Best Test Accuracy (p={best_p})')
plt.xlabel('p')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different p Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best p value: {best_p}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_p_index]}")


# ### 6.5. Spatial Algorithms Hyperparameter Tuning Curve

# Iterating KNN models with different spacial algorithm (auto, ball_tree, kd_tree, brute) using 10-fold cross-validation.

# In[27]:


algorithm_values = ['auto', 'ball_tree', 'kd_tree', 'brute']

train_accuracies = []
test_accuracies = []

for algorithm in algorithm_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=best_p, algorithm=algorithm)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
```

```
    test_accuracies.append(avg_test_accuracy)

best_algorithm_index, best_algorithm = find_best_value(test_accuracies, algorithm_values)

plt.plot(algorithm_values, train_accuracies, label='Average Training Accuracy')
plt.plot(algorithm_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_algorithm, test_accuracies[best_algorithm_index], color='red', marker='x', label=f'Best Test Accuracy (algorithm={best_algorithm})')
plt.xlabel('Algorithm')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Algorithm Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best algorithm: {best_algorithm}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_algorithm_index]}")


# ### 6.6. Leaf Size Values Hyperparameter Tuning Curve

# Iterating KNN models with different leaf size values (10 to 40) using 10-fold cross-validation.

# In[28]:


leaf_size_values = [10, 20, 30, 40]

train_accuracies = []
test_accuracies = []

for leaf_size in leaf_size_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=best_p, algorithm=best_algorithm, leaf_size=leaf_size)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_leaf_size_index, best_leaf_size = find_best_value(test_accuracies, leaf_size_values)

plt.plot(leaf_size_values, train_accuracies, label='Average Training Accuracy')
plt.plot(leaf_size_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_leaf_size, test_accuracies[best_leaf_size_index], color='red', marker='x', label=f'Best Test Accuracy (leaf_size={best_leaf_size})')
plt.xlabel('Leaf Size')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Leaf Size Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best leaf_size: {best_leaf_size}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_leaf_size_index]}")


# ### 6.7. Distance Metric Hyperparameter Tuning Curve

# Iterating KNN models with different distance metric (euclidean, manhattan, minkowski with best_p value) using 10-fold cross-validation.

# In[29]:


metric_values = ['euclidean', 'manhattan', 'minkowski']

train_accuracies = []
test_accuracies = []

kf = KFold(n_splits=10, shuffle=True, random_state=42)

for metric in metric_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=best_p, algorithm=best_algorithm, leaf_size=best_leaf_size, metric=metric)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_metric_index = np.argmax(test_accuracies)

best_metric = metric_values[best_metric_index]

best_metric_index, best_metric = find_best_value(test_accuracies, metric_values)

plt.plot(metric_values, train_accuracies, label='Average Training Accuracy')
plt.plot(metric_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_metric, test_accuracies[best_metric_index], color='red', marker='x', label=f'Best Test Accuracy (metric={best_metric})')
plt.xlabel('Metric')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Metric Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best metric: {best_metric}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_metric_index]}")


# ## 7. PCA Performance Trade-off

# Assessing PCA performance with varying number of components in a KNN model. It evaluates accuracy and records execution time, plotting results to evaluate trade-offs between accuracy

# In[48]:


import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score

n_components_values = list(range(1, 31))

accuracy_scores = []
execution_times = []

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

for n_components in n_components_values:
    pca = PCA(n_components=n_components)
    knn_pca = KNeighborsClassifier(
    algorithm=best_hyperparameters[0],
    leaf_size=best_hyperparameters[1],
    metric=best_hyperparameters[2],
    n_neighbors=best_hyperparameters[3],
    p=best_hyperparameters[4],
```

```
    weights=best_hyperparameters[5]
)
    pipeline = Pipeline([
        ('scaler', StandardScaler()),
        ('pca', pca),
        ('knn', knn_pca)
    ])

    start_time = time.time()

    scores = cross_val_score(pipeline, X_train, y_train, cv=kf, scoring='accuracy')

    accuracy_scores.append(np.mean(scores))
    execution_times.append(time.time() - start_time)

fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.set_xlabel('n_components')
ax1.set_ylabel('Accuracy', color=color)
ax1.plot(n_components_values, accuracy_scores, color=color, marker='o')
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()
color = 'tab:blue'
ax2.set_ylabel('Execution Time (seconds)', color=color)
ax2.plot(n_components_values, execution_times, color=color, marker='x')
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()
plt.title('PCA Performance for Different n_components Values')
plt.show()
```

4) Neural Network: Steps to run the code:

1) Run the file `main.py` in the directory Wisconsin/Neural_Network
2) Run the file `plot.py` in the directory Wisconsin/Neural_Network

Script plot.py in in the Directory Wisconsin/Neural_Network

```
import numpy as np
import matplotlib.pyplot as plt


def plot_array(arr1, label1, arr2, label2, title, xlabel, ylabel, fig_name):
    index = np.arange(1, len(arr1) + 1)
    plt.figure(figsize=(10, 6))
    plt.plot(index, arr1, '-o', color = 'orange' ,label=label1)
    plt.plot(index, arr2, '-o' ,label=label2)
    plt.legend()
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.grid(True)
    plt.savefig(fig_name)
    plt.show()


train_acc = np.load('train_acc_arr.npy')
val_acc = np.load('val_acc_arr.npy')
plot_array(train_acc, 'Training Accuracy', val_acc, 'Validation Accuracy', 'Training and Validation Accuracy vs Epochs', 'Epochs', 'Accuracy', 'accuracy_vs_epochs_fwisconsin.png')
```

Script main.py in in the Directory Wisconsin/Neural_Network

```
import ftns as ftns
import numpy as np
import torchvision
import torchvision.transforms as transforms
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import torch
from torch.utils.data import TensorDataset, DataLoader
import torch.nn as nn
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns



def main():
 X_train, y_train, X_test, y_test, X_val, y_val = ftns.load_fashion_dataset()

 X_train = np.load('X_train.npy')
 y_train = np.load('y_train.npy')
 X_test = np.load('X_test.npy')
 y_test = np.load('y_test.npy')
 X_val = np.load('X_val.npy')
 y_val = np.load('y_val.npy')

 print('X_train shape:', X_train.shape)
 print('y_train shape:', y_train.shape)
 print('X_val shape:', X_val.shape)
 print('y_val shape:', y_val.shape)
 print('X_test shape:', X_test.shape)
 print('y_test shape:', y_test.shape)
 print()


 train_data_tensor = torch.tensor(X_train, dtype=torch.float32)
 train_labs_tensor = torch.tensor(y_train, dtype=torch.long)
 train_dataset = TensorDataset(train_data_tensor, train_labs_tensor)

 val_data_tensor = torch.tensor(X_val, dtype=torch.float32)
 val_labs_tensor = torch.tensor(y_val, dtype=torch.long)
 val_dataset = TensorDataset(val_data_tensor, val_labs_tensor)

 test_data_tensor = torch.tensor(X_test, dtype=torch.float32)
 test_labs_tensor = torch.tensor(y_test, dtype=torch.long)
 test_dataset = TensorDataset(test_data_tensor, test_labs_tensor)

 batch_size = 64
 num_classes = 10
 learning_rate = 0.001
 num_epochs = 10

 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


 train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
 val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
 test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)



# Citation:
# Writing LeNet5 from Scratch in PyTorch, https://blog.paperspace.com/writing-lenet5-from-scratch-in-python/.



 class LeNet5(nn.Module):
  def __init__(self, num_classes):
   super(LeNet5, self).__init__()
   self.layer1 = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=0),
    nn.BatchNorm2d(6),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2))
   self.layer2 = nn.Sequential(
    nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
    nn.BatchNorm2d(16),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2))
   self.fc = nn.Linear(400, 120)
   self.relu = nn.ReLU()
   self.fully_conn1 = nn.Linear(120, 84)
   self.relu1 = nn.ReLU()
   self.fully_conn2 = nn.Linear(84, num_classes)

  def forward(self, x):
   out = self.layer1(x)
   out = self.layer2(out)
   out = out.view(out.size(0), -1)
   out = self.fc(out)
   out = self.relu(out)
   out = self.fully_conn1(out)
   out = self.relu1(out)
   out = self.fully_conn2(out)
   return out
```

```
model = LeNet5(num_classes).to(device)

criterion = nn.CrossEntropyLoss()

optim1 = torch.optim.Adam(model.parameters(), lr=learning_rate)


train_loss_lst = []
val_loss_lst = []
train_acc_lst = []
val_acc_lst = []


for epoch in range(num_epochs):
 train_all_loss = 0
 val_all_loss = 0

 train_fine = 0
 val_fine = 0

 train_all = 0
 val_all = 0

 for i, (inputs, labs) in enumerate(train_loader):
  inputs = inputs.to(device).unsqueeze(1)
  labs = labs.to(device)
  outputs = model(inputs)
  loss = criterion(outputs, labs)
  optim1.zero_grad()
  loss.backward()
  optim1.step()
  train_all_loss += loss.item()
  _, pred = torch.max(outputs.data, 1)
  train_all += labs.size(0)
  train_fine += (pred == labs).sum().item()



 with torch.no_grad():
  for i, (inputs, labs) in enumerate(val_loader):
   inputs = inputs.to(device).unsqueeze(1)
   labs = labs.to(device)
   outputs = model(inputs)
   loss = criterion(outputs, labs)
   val_all_loss += loss.item()
   _, pred = torch.max(outputs.data, 1)
   val_all += labs.size(0)
   val_fine += (pred == labs).sum().item()



 train_normalized_loss = train_all_loss / len(train_loader.dataset)
 train_loss_lst.append(train_normalized_loss)
 np.save('train_loss_arr.npy', np.array(train_loss_lst))
 val_normalized_loss = val_all_loss / len(val_loader.dataset)
 val_loss_lst.append(val_normalized_loss)
 np.save('val_loss_arr.npy', np.array(val_loss_lst))

 print('epoch', epoch)
 print('train_normalized_loss', train_normalized_loss)
 print('val_normalized_loss', val_normalized_loss)
 print('train accuracy', train_fine/train_all)
 print('Validation accuracy', val_fine/val_all)
 print()

 train_acc_lst.append(train_fine/train_all)
 np.save('train_acc_arr.npy', np.array(train_acc_lst))
 val_acc_lst.append(val_fine/val_all)
 np.save('val_acc_arr.npy', np.array(val_acc_lst))


all_labs = []
all_predictions = []


cv_mean_acc, cv_error_bar = ftns.cross_validate(train_loader, num_classes, num_epochs, batch_size, device, learning_rate, num_epochs, criterion)



with torch.no_grad():
 fine = 0
 all1 = 0
 for inputs, labs in test_loader:
  inputs = inputs.to(device).unsqueeze(1)
  labs = labs.to(device)
  outputs = model(inputs)
  _, pred = torch.max(outputs.data, 1)
  all1 += labs.size(0)
  fine += (pred == labs).sum().item()
  #Chatgpt prompt: How to plot a confusion matrix given labels and groundtruths from a for loop.
  all_labs.extend(labs.cpu().numpy())
  all_predictions.extend(pred.cpu().numpy())
```

```
  testing_accuracy =  fine / all1
  print('testing_accuracy', testing_accuracy)
  print()



 cm = confusion_matrix(all_labs, all_predictions)
 plt.figure(figsize=(10, 7))
 ax = sns.heatmap(cm, annot=True, fmt='g',  cbar=False)
 plt.title('Confusion Matrix')
 plt.xlabel('pred labs')
 plt.ylabel('True labs')
 plt.savefig('confusion_matrix.png')




 np.save('final testing accuracy', testing_accuracy)
 np.save('cv_mean_acc', cv_mean_acc)
 np.save('cv_error_bar', cv_error_bar )



 return 1

main()
```

Script ftns.py in the Directory Wisconsin/Neural_Network

```
import ftns as ftns
from ucimlrepo import fetch_ucirepo

import torch as t
import torch
import torch.cuda as cuda
import torch.optim as optimizer
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transf
from torchsummary import summary

from torch.utils.data import DataLoader

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt



def loading_dataset(shuffle):
 breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)
 X = breast_cancer_wisconsin_diagnostic.data.features
 y = breast_cancer_wisconsin_diagnostic.data.targets
 y['Diagnosis'] = y['Diagnosis'].map({'M': 1, 'B': 0})
 df_concatenated = pd.concat([X, y], axis=1)
 if shuffle == True:
  df_concatenated = df_concatenated.sample(frac=1).reset_index(drop=True)
 return df_concatenated



# Chatgpt prompt: How to split data into train, validation, and test sets by indices
def split_samples_with_indices_v2(total_samples, train_ratio=0.7, validation_ratio=0.2, test_ratio=0.1):
    train_samples = int(total_samples * train_ratio)
    validation_samples = int(total_samples * validation_ratio)
    test_samples = total_samples - train_samples - validation_samples
    train_start, train_end = 0, train_samples
    validation_start, validation_end = train_end, train_end + validation_samples
    test_start, test_end = validation_end, total_samples
    return train_samples, (train_start, train_end), validation_samples, (validation_start, validation_end), test_samples, (test_start, test_end)
```

Script data_feed.py in the Directory Wisconsin/Neural_Network

```
import os
import random
import pandas as pd
import torch
import numpy as np
import random
#from skimage import io
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, utils
import ast



def create_samples(X, samples_range):
    data_samples = []
    for idx, row in X[samples_range[0]:samples_range[1]].iterrows():
        data = list(row.values)
        data_samples.append(data)
    return data_samples



class DataFeed(Dataset):

    def __init__(self, X, samples_range):
        self.samples_range = samples_range
        self.samples = create_samples(X, samples_range)


    def __len__(self):
        return len( self.samples )


    def __getitem__(self, idx):
        sample = self.samples[idx]
        input_features = sample[:-1]
        output = sample[-1]
        input_arr = np.asarray(input_features)
        output_arr = np.asarray(output)


        return (input_arr, output_arr)
```

B. Adult Income Dataset

1) Logistic Regression: Steps to run the code:

1) Run the file SSHAH_TermProject_Adult_LogisticRegression_GD_Final.ipynb in the directory UCI/Logistic_Regression.
2) Run the file SSHAH_TermProject_Adult_LogisticRegression_SGD_Final.ipynb in the directory UCI/Logistic_Regression.

```
# -*- coding: utf-8 -*-
"""SSHAH_TermProject_Adult_LogisticRegression_GD_Final.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/15Ly3aWnCc-fzn2QvAbFUpP1qriJiFMc3

## Importing Data and Libraries
"""

!pip install ucimlrepo

from ucimlrepo import fetch_ucirepo
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
from sklearn.decomposition import PCA
from sklearn.model_selection import KFold

import matplotlib.pyplot as plt
import seaborn as sns
import torch
import torch.nn as nn

import copy
import time

torch.manual_seed(42)
np.random.seed(42)

adult_uci = fetch_ucirepo(id=2)

dataset = adult_uci.data.original
dataset['labels'] = dataset.income

for i in range(len(dataset.labels)):
  if (dataset.labels[i] == '<=50K.'):
    dataset.labels[i] = '<=50K'
  elif (dataset.labels[i] == '>50K.'):
    dataset.labels[i] = '>50K'

dataset = dataset.drop(columns=['income'])

print(dataset.labels[4881-1])
print(dataset.info())

"""## Pre-processing Data: Cleaning, Equalizing, and LabelEncoding"""

missing = dataset.isnull().sum()
duplicates = dataset.duplicated().sum()

dataset = dataset.dropna()
dataset = dataset.drop_duplicates()

print("Missing values removed:\n" + str(missing))
print("\nDuplicates removed:", duplicates)
print(dataset.info())

labelCount = 0
data = dataset
numerical = dataset.select_dtypes(include=['int64']).columns
categoricalColumns = dataset.select_dtypes(include=['object']).columns
categoricalColumns = [column for column in categoricalColumns if column != 'labels']
```

```
for column in categoricalColumns:
    labelCount += len(dataset[f'{column}'].unique())
    one_hot_encoded = pd.get_dummies(dataset[column], prefix=f'{column}_')
    data = pd.concat([data, one_hot_encoded], axis=1)

data = data.drop(columns=categoricalColumns)
data['labels'] = dataset['labels']

print(data.info(), labelCount, len(categoricalColumns), len(numerical))

classCount = data['labels'].value_counts()
minClassCount = classCount.min()
print(classCount)
newData = pd.DataFrame(columns=data.columns)

for label in classCount.index:
      classSamples = data[data['labels'] == label]
      newSamples = classSamples.sample(minClassCount)
      newData = pd.concat([newData, newSamples])

data = newData
print(data.info())

featuresRaw = data.drop('labels', axis=1)
labelsRaw = data.labels

labelMapping = {}

labelEncoder = LabelEncoder()
labelsEncoded = labelEncoder.fit_transform(labelsRaw)

for class_ in labelEncoder.classes_:
  encoding = labelEncoder.transform([class_])[0]
  labelMapping[encoding] = class_

labels = labelsEncoded
print(labelMapping)

"""## Splitting Data"""

X_train, X_test, y_train, y_test = train_test_split(featuresRaw, labels, test_size=0.3,
random_state=42)

trainScaler = StandardScaler()
X_train = trainScaler.fit_transform(X_train)
X_test = trainScaler.transform(X_test)

X_train_df = pd.DataFrame(X_train, columns=featuresRaw.columns)
X_test_df = pd.DataFrame(X_test, columns=featuresRaw.columns)

X_train_df.head()

X_data = np.array(featuresRaw)
y_data = np.array(labels)

"""## Model Construction and Training"""

class LogisticRegression(nn.Module):
  def __init__(self, num_features, l2=0.0):
    super(LogisticRegression, self).__init__()
    self.layer1 = nn.Linear(num_features, 1)
    self.sig = nn.Sigmoid()
    self.l2 = l2


  def forward(self, x):
    out0 = self.layer1(x)
```

```
    out1 = self.sig(out0)
    return out1

def resetWeights(model):
    for layer in model.children():
        if isinstance(layer, nn.Linear):
            layer.reset_parameters()

def trainModelKFold(model, epochs, lossFn, X, y, lr=0.01, batchSize=32, decisionThreshold=0.5,
k=1):
    trainAcc, testAcc = np.zeros(epochs), np.zeros(epochs)
    trainErr, testErr = np.zeros(epochs), np.zeros(epochs)
    trainLoss, testLoss = np.zeros(epochs), np.zeros(epochs)

    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        trainScaler = StandardScaler()
        X_train = trainScaler.fit_transform(X_train)
        X_test = trainScaler.transform(X_test)

        X_train_tensor = torch.tensor(X_train).float()
        X_test_tensor = torch.tensor(X_test).float()
        y_train_tensor = torch.tensor(y_train).float()
        y_test_tensor = torch.tensor(y_test).float()

        resetWeights(model)
        train_dataset = torch.utils.data.TensorDataset(X_train_tensor, y_train_tensor)
        train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
batch_size=batchSize, shuffle=True)
        for epoch in range(epochs):
            for batch_X, batch_y in train_loader:
              output = model(batch_X)
              loss = lossFn(output.squeeze(), batch_y)
              loss.backward()

              with torch.no_grad():
                  for param in model.parameters():
                      param.data -= lr * param.grad

                  model.zero_grad()

            with torch.no_grad():
                trainLoss[epoch] += loss.item()
                trainPredictions = (np.array(output.flatten()) > 0.5).astype(int)
                trainError = np.mean(trainPredictions != np.array(batch_y).astype(int))

                output = model(X_test_tensor)
                loss = lossFn(output.squeeze(), y_test_tensor)

                testLoss[epoch] += loss.item()
                testPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
                testError = np.mean(testPredictions != np.array(y_test_tensor).astype(int))

                trainErr[epoch] += trainError.item()
                testErr[epoch] += testError.item()
                trainAcc[epoch] += 1 - trainError.item()
                testAcc[epoch] += 1 - testError.item()

    avg_trainAcc = trainAcc/k
    avg_testAcc = testAcc/k
    avg_trainErr = trainErr/k
    avg_testErr = testErr/k
    avg_trainLoss = trainLoss/k
```

```
    avg_testLoss = testLoss/k

    return avg_trainAcc, avg_testAcc, avg_trainErr, avg_testErr, avg_trainLoss, avg_testLoss

def trainModel(model, epochs, lossFn, X_train, X_test, y_train, y_test, lr=0.01, batchSize=32,
decisionThreshold=0.5):
    trainAcc, testAcc = np.zeros(epochs), np.zeros(epochs)
    trainErr, testErr = np.zeros(epochs), np.zeros(epochs)
    trainLoss, testLoss = np.zeros(epochs), np.zeros(epochs)
    batchCount = 0

    resetWeights(model)
    train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
    train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batchSize,
shuffle=True)
    for epoch in range(epochs):
        for batch_X, batch_y in train_loader:
          batchCount += 1
          output = model(batch_X)
          loss = lossFn(output.squeeze(), batch_y)
          loss.backward()

          with torch.no_grad():
              for param in model.parameters():
                  param.data -= lr * param.grad

              model.zero_grad()

          with torch.no_grad():
              trainLoss[epoch] += loss.item()
              trainPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
              trainError = np.mean(trainPredictions != np.array(y_train).astype(int))

              output = model(X_test)
              loss = lossFn(output.squeeze(), y_test)

              testLoss[epoch] += loss.item()
              testPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
              testError = np.mean(testPredictions != np.array(y_test).astype(int))

              trainErr[epoch] += trainError.item()
              testErr[epoch] += testError.item()
              trainAcc[epoch] += 1 - trainError.item()
              testAcc[epoch] += 1 - testError.item()


    trainAcc = trainAcc/(batchCount/epochs)
    testAcc = testAcc/(batchCount/epochs)
    trainErr = trainErr/(batchCount/epochs)
    testErr = testErr/(batchCount/epochs)
    trainLoss = trainLoss/(batchCount/epochs)
    testLoss = testLoss/(batchCount/epochs)

    return trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss


X_train_tensor = torch.tensor(X_train).float()
y_train_tensor = torch.tensor(y_train).float()
X_test_tensor = torch.tensor(X_test).float()
y_test_tensor = torch.tensor(y_test).float()

"""## Grid Search Analysis"""

epochs_list = [50, 100, 250]
learning_rates = [1, 0.5, 0.05]
batchSizes = [1024]
lambdas = [0, 0.001, 0.1]
```

```
trainedModels = []

num_features = len(X_train_df.columns)
lossFn = nn.BCELoss()

for lambda_ in lambdas:
  model = LogisticRegression(num_features, l2=lambda_)
  for batchSize in batchSizes:
    for lr in learning_rates:
        for epochs in epochs_list:
          startTime = time.time()
          trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModelKFold(
              model,
              epochs,
              lossFn,
              X_data,
              y_data,
              lr=lr,
              batchSize=batchSize,
              k=5,
          )
          trainingTime = time.time() - startTime
          trainedModels.append({ 'model': copy.deepcopy(model),
                                 'hyperparameters': {'epochs': epochs, 'lr': lr, 'batchSize':
batchSize, 'lambda': lambda_ },
                                 'trainingStats': { 'testAcc': testAcc[-1], 'trainAcc':
trainAcc[-1],
                                                    'testError': testErr[-1], 'trainError':
trainErr[-1],
                                                    'testLoss': testLoss[-1], 'trainLoss':
trainLoss[-1] },
                                 'trainingTime': trainingTime
                                 })
          print(f'Completed epochs, lr, batchSize, lambda, test error: {epochs}, {lr},
{batchSize}, {lambda_}, --- {testAcc[-1]}')

"""## Find best model that maximizies accuracy using ROC curve:"""

def findOptimalThresholdForModelAcc(model, X_test_tensor, y_test_tensor):
  with torch.no_grad(): output = model(X_test_tensor)
  _, _, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
  optimalAcc = None
  optimalThreshold = None

  for threshold in thresholds:
    testPredictions = (np.array(output.detach().numpy().flatten()) > threshold).astype(int)
    testAcc = 1 - np.mean(testPredictions != np.array(y_test_tensor).astype(int))

    if (optimalThreshold == None):
      optimalThreshold = threshold
      optimalAcc = testAcc
    elif (testAcc > optimalAcc):
      optimalThreshold = threshold
      optimalAcc = testAcc



  return optimalAcc, optimalThreshold

globalOptimalModel = None
globalOptimalLoss = None
globalOptimalAcc = None
globalOptimalThreshold = None

for model_ in trainedModels:
    optimalAcc, optimalThreshold = findOptimalThresholdForModelAcc(
```

```
        model_['model'],
        X_test_tensor,
        y_test_tensor
    )
    optimalLoss = model_['trainingStats']['testLoss']

    if (globalOptimalLoss == None):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalAcc > globalOptimalAcc):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalAcc == globalOptimalAcc and optimalLoss < globalOptimalLoss):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold

print(globalOptimalModel)

gModel = globalOptimalModel['model']
with torch.no_grad(): output = gModel(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)

plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

gLr = globalOptimalModel['hyperparameters']['lr']
gEpochs = globalOptimalModel['hyperparameters']['epochs']
gBatchSize = globalOptimalModel['hyperparameters']['batchSize']
gLambda = globalOptimalModel['hyperparameters']['lambda']

index = int(np.where(thresholds == globalOptimalThreshold)[0])
FPR = fpr[index]
TPR = tpr[index]

print(f'Global Optimal Model Hyperparameters: lr={gLr}, epochs={gEpochs}, batchSize=
{gBatchSize}, lamda={gLambda}')
print(f'Global Optimal TPR: {TPR}, Global Optimal FPR: {FPR}, Global Optimal Threshold:
{globalOptimalThreshold}')
print(f'Global Optimal Acc: {globalOptimalAcc}, Global Optimal Loss: {globalOptimalLoss}')

"""### Reproducability Test & Visualization"""

lr = globalOptimalModel['hyperparameters']['lr']
epochs = globalOptimalModel['hyperparameters']['epochs']
batchSize = globalOptimalModel['hyperparameters']['batchSize']
lambda_ = globalOptimalModel['hyperparameters']['lambda']

num_features = len(X_train_df.columns)
model = LogisticRegression(num_features, l2=lambda_)
lossFn = nn.BCELoss()

trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModelKFold(
```

```
    model,
    epochs,
    lossFn,
    X_data,
    y_data,
    batchSize=batchSize,
    k=3,
)

model.eval()

plt.plot(trainAcc, label='train accuracy')
plt.plot(testAcc, label='test accuracy')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainErr, label='train error')
plt.plot(testErr, label='test error')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainLoss, label='train loss')
plt.plot(testLoss, label='test loss')
plt.grid(True)
plt.legend()

plt.show()


print(f'Train Accuracy: {trainAcc[-1]}, Train Error: {trainErr[-1]}, Train Loss: {trainLoss[-
1]}')
print(f'Test Accuracy: {testAcc[-1]}, Test Error: {testErr[-1]}, Test Loss: {testLoss[-1]}')

with torch.no_grad(): output = model(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)

plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

optimalAcc, optimalThreshold = findOptimalThresholdForModelAcc(
        model,
        X_test_tensor,
        y_test_tensor,
    )

index = int(np.where(thresholds == optimalThreshold)[0])
FPR = fpr[index]
TPR = tpr[index]

print(f'Global Optimal Model Hyperparameters: lr={lr}, epochs={epochs}, batchSize={batchSize},
lamda={lambda_}')
print(f'Global Optimal TPR: {TPR}, Global Optimal FPR: {FPR}, Global Optimal Threshold:
{optimalThreshold}')
print(f'Global Optimal Acc: {optimalAcc}, Global Optimal Loss: {testLoss[-1]}')

"""## Time Complexity Analysis"""
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

lambdaValue = 0.001
filteredModels = [modelInfo for modelInfo in trainedModels if modelInfo['hyperparameters']
['lambda'] == lambdaValue]

epochsList = [modelInfo['hyperparameters']['epochs'] for modelInfo in filteredModels]
learningRates = [modelInfo['hyperparameters']['lr'] for modelInfo in filteredModels]
batchSizes = [modelInfo['hyperparameters']['batchSize'] for modelInfo in filteredModels]
trainingTimes = [modelInfo['trainingTime'] for modelInfo in filteredModels]

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
sc = ax.scatter(epochsList, learningRates, batchSizes, c=trainingTimes, cmap='viridis',
marker='o', s=100)
ax.set_xlabel('Epochs')
ax.set_ylabel('Learning Rate')
ax.set_zlabel('Batch Size')
ax.set_title('Training Time Variation with Hyperparameters')
cbar = plt.colorbar(sc)
cbar.set_label('Training Time (seconds)')

plt.show()
```

```
# -*- coding: utf-8 -*-
"""SSHAH_TermProject_Adult_LogisticRegression_SGD_Final.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1wptHEGxBQxySDkAkGluVEcYUVlLvFz2A

# Importing Data and Libraries
"""

!pip install ucimlrepo

from ucimlrepo import fetch_ucirepo
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
from sklearn.decomposition import PCA
from sklearn.model_selection import KFold

import matplotlib.pyplot as plt
import seaborn as sns
import torch
import torch.nn as nn

import copy
import time

torch.manual_seed(42)
np.random.seed(42)

adult_uci = fetch_ucirepo(id=2)

dataset = adult_uci.data.original
dataset['labels'] = dataset.income

for i in range(len(dataset.labels)):
  if (dataset.labels[i] == '<=50K.'):
    dataset.labels[i] = '<=50K'
  elif (dataset.labels[i] == '>50K.'):
    dataset.labels[i] = '>50K'

dataset = dataset.drop(columns=['income'])

print(dataset.labels[4881-1])
print(dataset.info())

"""# Pre-processing Data: Cleaning, Equalizing, and LabelEncoding"""

missing = dataset.isnull().sum()
duplicates = dataset.duplicated().sum()

dataset = dataset.dropna()
dataset = dataset.drop_duplicates()

print("Missing values removed:\n" + str(missing))
print("\nDuplicates removed:", duplicates)
print(dataset.info())

labelCount = 0
data = dataset
numerical = dataset.select_dtypes(include=['int64']).columns
categoricalColumns = dataset.select_dtypes(include=['object']).columns
categoricalColumns = [column for column in categoricalColumns if column != 'labels']
```

```
for column in categoricalColumns:
    labelCount += len(dataset[f'{column}'].unique())
    one_hot_encoded = pd.get_dummies(dataset[column], prefix=f'{column}_')
    data = pd.concat([data, one_hot_encoded], axis=1)

data = data.drop(columns=categoricalColumns)
data['labels'] = dataset['labels']

print(data.info(), labelCount, len(categoricalColumns), len(numerical))

classCount = data['labels'].value_counts()
minClassCount = classCount.min()
print(classCount)
newData = pd.DataFrame(columns=data.columns)

for label in classCount.index:
      classSamples = data[data['labels'] == label]
      newSamples = classSamples.sample(minClassCount)
      newData = pd.concat([newData, newSamples])

data = newData
print(data.info())

featuresRaw = data.drop('labels', axis=1)
labelsRaw = data.labels

labelMapping = {}

labelEncoder = LabelEncoder()
labelsEncoded = labelEncoder.fit_transform(labelsRaw)

for class_ in labelEncoder.classes_:
  encoding = labelEncoder.transform([class_])[0]
  labelMapping[encoding] = class_

labels = labelsEncoded
print(labelMapping)

"""# Splitting Data"""

X_train, X_test, y_train, y_test = train_test_split(featuresRaw, labels, test_size=0.3,
random_state=42)

trainScaler = StandardScaler()
X_train = trainScaler.fit_transform(X_train)
X_test = trainScaler.transform(X_test)

X_train_df = pd.DataFrame(X_train, columns=featuresRaw.columns)
X_test_df = pd.DataFrame(X_test, columns=featuresRaw.columns)

X_train_df.head()

X_data = np.array(featuresRaw)
y_data = np.array(labels)

"""# Model Construction and Training"""

class LogisticRegression(nn.Module):
  def __init__(self, num_features, l2=0.0):
    super(LogisticRegression, self).__init__()
    self.layer1 = nn.Linear(num_features, 1)
    self.sig = nn.Sigmoid()
    self.l2 = l2


  def forward(self, x):
    out0 = self.layer1(x)
```

```
    out1 = self.sig(out0)
    return out1

def resetWeights(model):
    for layer in model.children():
        if isinstance(layer, nn.Linear):
            layer.reset_parameters()

def trainModelKFold(model, epochs, lossFn, optimizer, X, y, decisionThreshold=0.5, k=1):
    trainAcc, testAcc = np.zeros(epochs), np.zeros(epochs)
    trainErr, testErr = np.zeros(epochs), np.zeros(epochs)
    trainLoss, testLoss = np.zeros(epochs), np.zeros(epochs)

    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        trainScaler = StandardScaler()
        X_train = trainScaler.fit_transform(X_train)
        X_test = trainScaler.transform(X_test)

        X_train_tensor = torch.tensor(X_train).float()
        X_test_tensor = torch.tensor(X_test).float()
        y_train_tensor = torch.tensor(y_train).float()
        y_test_tensor = torch.tensor(y_test).float()

        resetWeights(model)
        for epoch in range(epochs):
            output = model(X_train_tensor)
            loss = lossFn(output.squeeze(), y_train_tensor)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            with torch.no_grad():
                trainLoss[epoch] += loss.item()
                trainPredictions = (np.array(output.flatten()) >
decisionThreshold).astype(int)
                trainError = np.mean(trainPredictions != np.array(y_train_tensor).astype(int))


                output = model(X_test_tensor)
                loss = lossFn(output.squeeze(), y_test_tensor)

                testLoss[epoch] += loss.item()
                testPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
                testError = np.mean(testPredictions != np.array(y_test_tensor).astype(int))

                trainErr[epoch] += trainError.item()
                testErr[epoch] += testError.item()
                trainAcc[epoch] += 1 - trainError.item()
                testAcc[epoch] += 1 - testError.item()

    avg_trainAcc = trainAcc/k
    avg_testAcc = testAcc/k
    avg_trainErr = trainErr/k
    avg_testErr = testErr/k
    avg_trainLoss = trainLoss/k
    avg_testLoss = testLoss/k

    return avg_trainAcc, avg_testAcc, avg_trainErr, avg_testErr, avg_trainLoss, avg_testLoss

def trainModel(model, epochs, lossFn, optimizer, X_train, X_test, y_train, y_test,
decisionThreshold=0.5):
    trainAcc, testAcc = [], []
```

```
    trainErr, testErr = [], []
    trainLoss, testLoss = [], []

    resetWeights(model)
    for epoch in range(epochs):
        output = model(X_train)
        loss = lossFn(output.squeeze(), y_train)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        with torch.no_grad():
            trainLoss.append(loss.item())
            trainPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
            trainError = np.mean(trainPredictions != np.array(y_train).astype(int))


            output = model(X_test)
            loss = lossFn(output.squeeze(), y_test)

            testLoss.append(loss.item())
            testPredictions = (np.array(output.flatten()) > decisionThreshold).astype(int)
            testError = np.mean(testPredictions != np.array(y_test).astype(int))

            trainErr.append(trainError.item())
            testErr.append(testError.item())
            trainAcc.append(1 - trainError.item())
            testAcc.append(1 - testError.item())

    return trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss


X_train_tensor = torch.tensor(X_train).float()
y_train_tensor = torch.tensor(y_train).float()
X_test_tensor = torch.tensor(X_test).float()
y_test_tensor = torch.tensor(y_test).float()

"""# Grid Search Analysis"""

epochs_list = [50, 100, 500, 1000]
learning_rates = [1, 0.1, 0.01, 0.001]
momentums = [0, 0.5, 0.9, 0.99]
lambdas = [0, 0.001, 0.01, 0.1, 1]
trainedModels = []

num_features = len(X_train_df.columns)
lossFn = nn.BCELoss()

for lambda_ in lambdas:
  model = LogisticRegression(num_features, l2=lambda_)
  for momentum in momentums:
    for lr in learning_rates:
        optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum,
weight_decay=model.l2)
        for epochs in epochs_list:
          startTime = time.time()
          trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModelKFold(
              model,
              epochs,
              lossFn,
              optimizer,
              X_data,
              y_data,
              k=5,
          )
          trainingTime = time.time() - startTime
          trainedModels.append({ 'model': copy.deepcopy(model),
```

```
                                'hyperparameters': {'epochs': epochs, 'lr': lr, 'momentum':
momentum, 'lambda': lambda_ },
                                'trainingStats': { 'testAcc': testAcc[-1], 'trainAcc':
trainAcc[-1],
                                                   'testError': testErr[-1], 'trainError':
trainErr[-1],
                                                   'testLoss': testLoss[-1], 'trainLoss':
trainLoss[-1] },
                                'trainingTime': trainingTime
                                })
          print(f'Completed epochs, lr, momentum, lambda, test error: {epochs}, {lr},
{momentum}, {lambda_}, --- {testAcc[-1]}')

"""# Finding the best model that maximizes accuracy using ROC curve:


"""

def findOptimalThresholdForModelAcc(model, X_test_tensor, y_test_tensor):
  with torch.no_grad(): output = model(X_test_tensor)
  _, _, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
  optimalAcc = None
  optimalThreshold = None

  for threshold in thresholds:
    testPredictions = (np.array(output.detach().numpy().flatten()) > threshold).astype(int)
    testAcc = 1 - np.mean(testPredictions != np.array(y_test_tensor).astype(int))
    if (optimalThreshold == None):
      optimalThreshold = threshold
      optimalAcc = testAcc
    elif (testAcc > optimalAcc):
      optimalThreshold = threshold
      optimalAcc = testAcc



  return optimalAcc, optimalThreshold

globalOptimalModel = None
globalOptimalLoss = None
globalOptimalAcc = None
globalOptimalThreshold = None

for model_ in trainedModels:
    optimalAcc, optimalThreshold = findOptimalThresholdForModelAcc(
        model_['model'],
        X_test_tensor,
        y_test_tensor
    )
    optimalLoss = model_['trainingStats']['testLoss']

    if (globalOptimalLoss == None):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalAcc > globalOptimalAcc):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
    elif (optimalAcc == globalOptimalAcc and optimalLoss < globalOptimalLoss):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
      globalOptimalThreshold = optimalThreshold
```

```
print(globalOptimalModel)

gModel = globalOptimalModel['model']
with torch.no_grad(): output = gModel(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)

plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

gLr = globalOptimalModel['hyperparameters']['lr']
gEpochs = globalOptimalModel['hyperparameters']['epochs']
gMomentum = globalOptimalModel['hyperparameters']['momentum']
gLambda = globalOptimalModel['hyperparameters']['lambda']

index = int(np.where(thresholds == globalOptimalThreshold)[0])
FPR = fpr[index]
TPR = tpr[index]

print(f'Global Optimal Model Hyperparameters: lr={gLr}, epochs={gEpochs}, momentum={gMomentum},
lamda={gLambda}')
print(f'Global Optimal TPR: {TPR}, Global Optimal FPR: {FPR}, Global Optimal Threshold:
{globalOptimalThreshold}')
print(f'Global Optimal Acc: {globalOptimalAcc}, Global Optimal Loss: {globalOptimalLoss}')

"""## Reproducability Test & Visualizing Training Curves"""

lr = globalOptimalModel['hyperparameters']['lr']
epochs = globalOptimalModel['hyperparameters']['epochs']
momentum = globalOptimalModel['hyperparameters']['momentum']
lambda_ = globalOptimalModel['hyperparameters']['lambda']

num_features = len(X_train_df.columns)
model = LogisticRegression(num_features, l2=lambda_)
lossFn = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum,
weight_decay=model.l2)

trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModelKFold(
    model,
    epochs,
    lossFn,
    optimizer,
    X_data,
    y_data,
    k=5,
)

model.eval()

plt.plot(trainAcc, label='train accuracy')
plt.plot(testAcc, label='test accuracy')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainErr, label='train error')
plt.plot(testErr, label='test error')
```

```
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainLoss, label='train loss')
plt.plot(testLoss, label='test loss')
plt.grid(True)
plt.legend()

plt.show()


print(f'Train Accuracy: {trainAcc[-1]}, Train Error: {trainErr[-1]}, Train Loss: {trainLoss[-
1]}')
print(f'Test Accuracy: {testAcc[-1]}, Test Error: {testErr[-1]}, Test Loss: {testLoss[-1]}')

with torch.no_grad(): output = model(X_test_tensor)
fpr, tpr, thresholds = roc_curve(y_test_tensor.detach().numpy(),
output.detach().numpy().flatten(), pos_label=1)
rocAUC = auc(fpr, tpr)

plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.format(rocAUC))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

optimalAcc, optimalThreshold = findOptimalThresholdForModelAcc(
        model,
        X_test_tensor,
        y_test_tensor,
    )

index = int(np.where(thresholds == optimalThreshold)[0])
FPR = fpr[index]
TPR = tpr[index]

print(f'Global Optimal Model Hyperparameters: lr={lr}, epochs={epochs}, momentum={momentum},
lamda={lambda_}')
print(f'Global Optimal TPR: {TPR}, Global Optimal FPR: {FPR}, Global Optimal Threshold:
{optimalThreshold}')
print(f'Global Optimal Acc: {optimalAcc}, Global Optimal Loss: {testLoss[-1]}')

"""# Time Complexity Analysis"""

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

lambdaValue = 0.001
filteredModels = [modelInfo for modelInfo in trainedModels if modelInfo['hyperparameters']
['lambda'] == lambdaValue]

epochsList = [modelInfo['hyperparameters']['epochs'] for modelInfo in filteredModels]
learningRates = [modelInfo['hyperparameters']['lr'] for modelInfo in filteredModels]
momentums = [modelInfo['hyperparameters']['momentum'] for modelInfo in filteredModels]
trainingTimes = [modelInfo['trainingTime'] for modelInfo in filteredModels]

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
sc = ax.scatter(epochsList, learningRates, momentums, c=trainingTimes, cmap='viridis',
marker='o', s=100)
ax.set_xlabel('Epochs')
ax.set_ylabel('Learning Rate')
```

```
ax.set_zlabel('Momentum')
ax.set_title('Training Time Variation with Hyperparameters')
cbar = plt.colorbar(sc)
cbar.set_label('Training Time (seconds)')

plt.show()
```

2) Support Vector Machine: Steps to run the code:

1) Run the file `UCI_SVM.ipynb` in the directory UCI/SVM. For running the file, the user just have to execute the "Run All Below" command, mentioned in the dropdown menu of the Cell tab in the Jupyter notebook.

```
# -*- coding: utf-8 -*-
"""UCI_SVM.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1pmeujPPcSz7EIwyWwdDCiMfEvXg_PZDy
"""

!pip3 install ucimlrepo

from ucimlrepo import fetch_ucirepo
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import validation_curve
from sklearn.model_selection import learning_curve
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import RandomizedSearchCV
from sklearn.decomposition import PCA
from scipy.stats import randint
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd

# Extracting the raw data
adult = fetch_ucirepo(id=2)

X = adult.data.features
y = adult.data.targets

print(adult.metadata)

print(adult.variables)

# Processing the dataset
unique_values = y['income'].unique()
print("Unique values in 'Category' column:", unique_values)

y['income'] = y['income'].replace({'>50K.': '>50K', '<=50K.': '<=50K'})

merged_df = pd.merge(X, y, left_index=True, right_index=True)
merged_df.head()

merged_df.replace("?", np.nan, inplace = True)

null_values = merged_df.isnull().sum()

print(null_values[null_values>0])

columns_of_interest = null_values[null_values > 0].index.tolist()

for column_name in columns_of_interest:
    missing_percentage = (merged_df[column_name].isnull().sum() / len(merged_df)) * 100
    print(f"Percentage of missing data in '{column_name}': {missing_percentage:.2f}%")

merged_df.dropna(inplace=True)

columns_of_interest = null_values[null_values > 0].index.tolist()

for column_name in columns_of_interest:
    missing_percentage = (merged_df[column_name].isnull().sum() / len(merged_df)) * 100
    print(f"Percentage of missing data in '{column_name}': {missing_percentage:.2f}%")
```

```
categorical_columns = merged_df.select_dtypes(include=['object']).columns

label_encoder_dict = {}

for col in categorical_columns:
    label_encoder = LabelEncoder()
    merged_df[col] = label_encoder.fit_transform(merged_df[col])
    label_encoder_dict[col] = dict(zip(label_encoder.classes_,
label_encoder.transform(label_encoder.classes_)))

from sklearn.preprocessing import StandardScaler

# Assuming 'income' is still present in the DataFrame
X = merged_df.drop('income', axis=1)
y = merged_df['income']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("\nScaled Training Data:")
print(pd.DataFrame(X_train_scaled, columns=X.columns).head())

# Performing PCA before SVM

num_components = 9
pca = PCA(n_components=num_components)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)
explained_variance_ratio = pca.explained_variance_ratio_
print("Explained Variance Ratios:", explained_variance_ratio)

cumulative_variance = explained_variance_ratio.cumsum()

total_explained_variance_percentage = cumulative_variance[-1] * 100
print(f"Total Explained Variance Percentage: {total_explained_variance_percentage:.2f}%")

# Performing SVM without kernel

# Defining the hyperparameter
param_grid_linear = {
    'C': randint(1, 11)
}

svm_linear = SVC(kernel='linear')

random_search_linear = RandomizedSearchCV(estimator=svm_linear,
param_distributions=param_grid_linear,
                                          n_iter=5, cv=5, scoring='accuracy', verbose=2,
n_jobs=-1)
random_search_linear.fit(X_train_pca, y_train)

best_svm_linear = random_search_linear.best_estimator_

y_pred_linear = best_svm_linear.predict(X_test_pca)

# Evaluating the performance of the linear SVM
accuracy_linear = accuracy_score(y_test, y_pred_linear)
print("Best Linear SVM Accuracy:", accuracy_linear)
print("Classification Report (Best Linear SVM):")
print(classification_report(y_test, y_pred_linear))
best_C_linear = random_search_linear.best_params_['C']
print("Best C:", best_C_linear)

C_values = list(range(1, 11))
```

```
train_scores, test_scores = [], []

for C in C_values:
    best_svm_linear.C = C
    best_svm_linear.fit(X_train_pca, y_train)

    train_score = best_svm_linear.score(X_train_pca, y_train)
    train_scores.append(train_score)

    test_score = best_svm_linear.score(X_test_pca, y_test)
    test_scores.append(test_score)

# Plotting the training and test scores against the hyperparameter C
plt.figure(figsize=(10, 6))
plt.plot(C_values, train_scores, 'o-', label='Training Score')
plt.plot(C_values, test_scores, 'o-', label='Test Score')
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Test Scores vs C')
plt.show()

# Plotting the learning curve
def plot_learning_curve(estimator, X, y, cv, train_sizes=np.linspace(0.1, 1.0, 3)):
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, train_sizes=train_sizes, scoring='accuracy', n_jobs=-1
    )

    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_scores_mean, 'o-', color='r', label='Training Score')
    plt.plot(train_sizes, test_scores_mean, 'o-', color='g', label='Validation Score')

    plt.title('Learning Curve')
    plt.xlabel('Training Examples')
    plt.ylabel('Score')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()

plot_learning_curve(best_svm_linear, X_train_pca, y_train, cv=5)

# Performing SVM with Polynomial Kernel

# Defining the hyperparameters
param_dist_poly = {
    'C': randint(4, 7),
    'degree': np.random.choice([1, 2, 3, 5, 10], size=1)
}

svm_poly = SVC(kernel='poly')

random_search_poly = RandomizedSearchCV(estimator=svm_poly,
param_distributions=param_dist_poly,
                                        n_iter=5, cv=5, scoring='accuracy', verbose=2,
n_jobs=-1)
random_search_poly.fit(X_train_pca, y_train)

best_svm_poly = random_search_poly.best_estimator_

y_pred_poly = best_svm_poly.predict(X_test_pca)

# Evaluating the performance of Polynomial SVM
```

```
accuracy_poly = accuracy_score(y_test, y_pred_poly)
print("Best Polynomial SVM Accuracy:", accuracy_poly)
print("Classification Report (Best Polynomial SVM):")
print(classification_report(y_test, y_pred_poly))
best_degree = random_search_poly.best_params_['degree']
print("Best Degree:", best_degree)
best_C_poly = random_search_poly.best_params_['C']
print("Best C:", best_C_poly)


# Plotting the training and test scores against polynomial degree
degrees = [1, 2, 3, 5, 10]
train_scores, test_scores = [], []

for degree in degrees:
    best_svm_poly.degree = degree
    best_svm_poly.fit(X_train_pca, y_train)

    train_score = best_svm_poly.score(X_train_pca, y_train)
    train_scores.append(train_score)

    test_score = best_svm_poly.score(X_test_pca, y_test)
    test_scores.append(test_score)

plt.figure(figsize=(10, 6))
plt.plot(degrees, train_scores, 'o-', label='Training Score')
plt.plot(degrees, test_scores, 'o-', label='Test Score')
plt.xlabel('Degree of polynomial')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Test Scores vs Degree for Polynomial SVM')
plt.show()

# Plotting the learning curve
plot_learning_curve(best_svm_poly, X_train_pca, y_train, cv=5)

# Performing SVM with RBF Kernel

# Defining the hyperparameters
param_grid_rbf = {
    'C': randint(4, 7),
    'gamma': np.random.choice([0.3, 0.4, 0.5, 0.6, 0.7], size=1)
}

svm_rbf = SVC(kernel='rbf')

random_search_rbf = RandomizedSearchCV(estimator=svm_rbf, param_distributions=param_grid_rbf,
                                       n_iter=5, cv=5, scoring='accuracy', verbose=2,
n_jobs=-1)
random_search_rbf.fit(X_train_pca, y_train)

best_svm_rbf = random_search_rbf.best_estimator_

y_pred_rbf = best_svm_rbf.predict(X_test_pca)

# Evaluating the performance of RBF SVM
accuracy_rbf = accuracy_score(y_test, y_pred_rbf)
print("Best RBF SVM Accuracy:", accuracy_rbf)
print("Classification Report (Best RBF SVM):")
print(classification_report(y_test, y_pred_rbf))
best_gamma = random_search_rbf.best_params_['gamma']
print("Best Gamma:", best_gamma)
best_C_rbf = random_search_rbf.best_params_['C']
print("Best C:", best_C_rbf)

gammas = [0.1, 0.3, 0.5, 0.7]
train_scores, test_scores = [], []
```

```
for gamma in gammas:
    best_svm_rbf.gamma = gamma
    best_svm_rbf.fit(X_train_pca, y_train)

    train_score = best_svm_rbf.score(X_train_pca, y_train)
    train_scores.append(train_score)

    test_score = best_svm_rbf.score(X_test_pca, y_test)
    test_scores.append(test_score)

# Plotting the training and test scores against gamma
plt.figure(figsize=(10, 6))
plt.plot(gammas, train_scores, 'o-', label='Training Score')
plt.plot(gammas, test_scores, 'o-', label='Test Score')
plt.xlabel('Gamma of RBF')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Test Scores vs Gamma for RBF SVM')
plt.show()

# Plotting the learning curve
plot_learning_curve(best_svm_rbf, X_train_pca, y_train, cv=5)

# Computing the confusion matrix for the selected SVM model

from sklearn.metrics import confusion_matrix

conf_matrix = confusion_matrix(y_test, y_pred_rbf)

# Plotting the confusion matrix

plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False,
            xticklabels=best_svm_rbf.classes_, yticklabels=best_svm_rbf.classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

3) k-Nearest Neighbours: Steps to run the code:

1) Run the file `TMahmood_TermProject_Adult_kNN.ipynb` in the directory UCI/kNN. For running the file, the user just have to execute the "Run All Below" command, mentioned in the dropdown menu of the Cell tab in the Jupyter notebook.

```
#!/usr/bin/env python
# coding: utf-8

# ## 1. Importing Libraries

# In[1]:


pip install ucimlrepo


# In[1]:


from ucimlrepo import fetch_ucirepo
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import KFold
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler


# ## 2. Importing Data

# In[3]:


from ucimlrepo import fetch_ucirepo

adult = fetch_ucirepo(id=2)

X = adult.data.features
y = adult.data.targets


# ## 3. Data Preprocessing

# Standardizing entries ('>50K.' to '>50K', '<=50K.' to '<=50K') for consistency in the DataFrame 'y'.

# In[4]:


unique_values = y['income'].unique()
print("Unique values in 'Category' column:", unique_values)
y['income'] = y['income'].replace({'>50K.': '>50K', '<=50K.': '<=50K'})


# ### 3.1. Checking Data set Type

# Examining the data types of each feature in the dataset to ensure the absence of data types that could potentially impact performance adversely.

# In[6]:


print("Data Types:")
print(X.dtypes)

print("Target Data Types:")
print(y.dtypes)


# In[7]:


merged_df = pd.merge(X, y, left_index=True, right_index=True)

merged_df.head()


# Generating histograms for categorical columns in the entire dataset.

# In[44]:


features = merged_df.select_dtypes(include='object').columns

for feature in features:
    plt.figure(figsize=(12, 6))
    sns.countplot(x=feature, hue='income', data=merged_df)
    plt.title(f'{feature} vs Income')
    plt.xlabel(feature)
    plt.ylabel('Count')
    plt.show()


# Generating histograms with kernel density estimates for numerical columns in the entire dataset.

# In[10]:


numerical_features = merged_df.select_dtypes(include=['int64']).columns

for feature in numerical_features:
    plt.figure(figsize=(12, 6))
    sns.histplot(data=merged_df, x=feature, hue='income', bins=30, kde=True)
    plt.title(f'{feature} vs Income')
    plt.xlabel(feature)
    plt.ylabel('Count')
    plt.show()


# ### 3.2. Checking and removing Null Values

# Processing the data types of each feature in the dataset to ensure the absence of data types that could potentially impact performance adversely.

# In[11]:


merged_df.replace("?", np.nan, inplace = True)

null_values = merged_df.isnull().sum()
```

```
print(null_values[null_values>0])


# Null values before removal

# In[12]:


columns_of_interest = null_values[null_values > 0].index.tolist()

for column_name in columns_of_interest:
    missing_percentage = (merged_df[column_name].isnull().sum() / len(merged_df)) * 100
    print(f"Percentage of missing data in '{column_name}': {missing_percentage:.2f}%")


# Null values after removal

# In[13]:


merged_df.dropna(inplace=True)
merged_df.reset_index(drop=True, inplace=True)

columns_of_interest = null_values[null_values > 0].index.tolist()

for column_name in columns_of_interest:
    missing_percentage = (merged_df[column_name].isnull().sum() / len(merged_df)) * 100
    print(f"Percentage of missing data in '{column_name}': {missing_percentage:.2f}%")


# ### 3.3. Label Encoding

# Applying Label Encoding to transform categorical features into numerical format for better machine learning model training.

# In[15]:


categorical_columns = merged_df.select_dtypes(include=['object']).columns

label_encoder_dict = {}

for col in categorical_columns:
    label_encoder = LabelEncoder()
    merged_df[col] = label_encoder.fit_transform(merged_df[col])
    label_encoder_dict[col] = dict(zip(label_encoder.classes_, label_encoder.transform(label_encoder.classes_)))


# ### 3.4. Scalarization of the Dataset

# Applying standardization to the feature matrix (X) using StandardScaler, creating X_scaled_df as a DataFrame. This ensures consistent scaling, aiding machine learning model performan

# In[16]:


X = merged_df.drop('income', axis=1)
y = merged_df['income']

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)

X_scaled_df.head()


# ### 3.5. Principal Component Analysis

# Using Principal Component Analysis (PCA) on the scaled features (X_scaled_df), visualizing cumulative explained variance of principal components for dimensionality reduction.

# In[19]:


pca = PCA()
X_pca = pca.fit_transform(X_scaled)

explained_variance_ratio = pca.explained_variance_ratio_

cumulative_variance = np.cumsum(explained_variance_ratio)

plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, marker='o', linestyle='--')
plt.title('Cumulative Explained Variance')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Variance Explained')
plt.grid(True)
plt.savefig("Variance")


# Visualizing the explained variance ratio for each principal component using a bar plot and cumulative step plot.

# In[20]:


plt.bar(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio, alpha=0.75, align='center')
plt.step(range(1, len(explained_variance_ratio) + 1), np.cumsum(explained_variance_ratio), where='mid')
plt.title('Explained Variance Ratio for Each Principal Component')
plt.xlabel('Principal Component Number')
plt.ylabel('Explained Variance Ratio')
plt.savefig("Variance_Ratio")


# Creating a bar plot to visualize the absolute loadings of Principal Component 1 (PC1) in a PCA-transformed dataset, highlighting key features and their contributions to the variance

# In[21]:


feature_names = X.columns.tolist()

loadings_df = pd.DataFrame(pca.components_, columns=feature_names)

loadings_df = loadings_df[feature_names]

sorted_feature_names = loadings_df.abs().iloc[0, :].sort_values(ascending=False).index
sorted_loadings = loadings_df[sorted_feature_names]

plt.bar(sorted_feature_names, np.abs(sorted_loadings.iloc[0, :]))
plt.title('Absolute Loadings for PC1')
plt.xlabel('Features')
plt.ylabel('Absolute Loading')
plt.xticks(rotation=45, ha='right')
```

```
plt.savefig("Absolute_Loading")


# Performing PCA to retain 90% of the variance in the scaled dataset.

# In[22]:


cumulative_variance_ratio = np.cumsum(pca.explained_variance_ratio_)
n_components_90 = np.argmax(cumulative_variance_ratio >= 0.9) + 1

pca = PCA(n_components=n_components_90)
X_pca = pca.fit_transform(X_scaled_df)

columns_pca = [f"PC_{i+1}" for i in range(X_pca.shape[1])]
pca_df = pd.DataFrame(data=X_pca, columns=columns_pca)
pca_df.head()


# ## 4. Model Training (Grid Search)

# ### 4.1. Defining the model and hyperparameters

# Defining the function to initiate kFold and knn, and assiging the hyperparameter range in the param_grid dictionary.

# In[24]:


y = merged_df['income']
X_train, X_test, y_train, y_test = train_test_split(pca_df, y, test_size=0.2, random_state=42)
knn = KNeighborsClassifier()
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11, 13, 15],
    'weights': ['uniform', 'distance'],
    'p': [i/2 for i in range(2, 9)],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'leaf_size': [10, 20, 30, 40],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}
kf = KFold(n_splits=10, shuffle=True, random_state=42)


# In[33]:


grid_search = GridSearchCV(knn, param_grid, cv=kf, scoring='accuracy')

results = grid_search.fit(X_train, y_train)


# ### 4.2. Determing the best model based on validation accuracy

# Examining different hyperparameter combinations and their accuracies through a grid search. The bar plot displays the top 5 configurations with their accuracy scores, helping identif

# In[47]:


hyperparameters = []
accuracies = []

for params, mean_score, _ in zip(grid_search.cv_results_['params'], grid_search.cv_results_['mean_test_score'], grid_search.cv_results_['std_test_score']):
    params_tuple = tuple(params.values())

    hyperparameters.append(params_tuple)
    accuracies.append(mean_score)

results_df = pd.DataFrame({'Hyperparameters': hyperparameters, 'Accuracy': accuracies})

results_df = results_df.sort_values(by='Accuracy', ascending=False)

top_n = 5
plt.figure(figsize=(12, 6))
ax = sns.barplot(x='Accuracy', y='Hyperparameters', data=results_df.head(top_n), orient='h')

for index, value in enumerate(results_df.head(top_n)['Accuracy']):
    ax.text(value, index, f'{value:.3f}', ha='left', va='center', color='black')

ax.set_xlim(0, 1.1 * max(results_df['Accuracy']))

plt.title(f'Top {top_n} Accuracy Across Different Hyperparameter Values')
plt.xlabel('Accuracy')
plt.ylabel('Hyperparameters')
plt.show()


# In[38]:


best_hyperparameters = results_df.iloc[0]['Hyperparameters']

print(f"Best Hyperparameters: "
      f"algorithm={best_hyperparameters[0]}, "
      f"leaf_size={best_hyperparameters[1]}, "
      f"metric={best_hyperparameters[2]}, "
      f"n_neighbors={best_hyperparameters[3]}, "
      f"p={best_hyperparameters[4]}, "
      f"weights={best_hyperparameters[5]}")


# ### 4.3. Examining the performance of the best model

# Using the best hyperparameters obtained from a grid search, metrics such as accuracy, precision, recall, and F1 score are computed and presented, along with a heatmap visualization o

# In[40]:


best_knn = KNeighborsClassifier(
    algorithm=best_hyperparameters[0],
    leaf_size=best_hyperparameters[1],
    metric=best_hyperparameters[2],
    n_neighbors=best_hyperparameters[3],
    p=best_hyperparameters[4],
    weights=best_hyperparameters[5]
)

best_knn.fit(X_train, y_train)

y_pred = best_knn.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
```

```
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

print(f'Best Model Accuracy on Test Set: {accuracy:.4f}')
print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1 Score: {f1:.4f}')

conf_matrix = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(4, 3))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()


# ## 5. Model Training (Iterative Search)

# ### 5.1. Calling the helper functions for model training

# Defining functions for KNN cross-validation, training on folds, and finding optimal hyperparameter values through averaging test accuracies.

# In[25]:


def knn_cross_val_fold(X_train_fold, y_train_fold, X_val_fold, y_val_fold, knn):
    knn.fit(X_train_fold, y_train_fold)

    y_train_fold_pred = knn.predict(X_train_fold)
    y_val_fold_pred = knn.predict(X_val_fold)

    fold_train_accuracy = accuracy_score(y_train_fold, y_train_fold_pred)
    fold_test_accuracy = accuracy_score(y_val_fold, y_val_fold_pred)

    return fold_train_accuracy, fold_test_accuracy

def knn_cross_val(X_train, y_train, knn, kf=kf):
    fold_train_accuracies = []
    fold_test_accuracies = []

    for train_index, test_index in kf.split(X_train):
        X_train_fold, X_val_fold = X_train.iloc[train_index], X_train.iloc[test_index]
        y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.iloc[test_index]

        fold_train_accuracy, fold_test_accuracy = knn_cross_val_fold(X_train_fold, y_train_fold, X_val_fold, y_val_fold, knn)

        fold_train_accuracies.append(fold_train_accuracy)
        fold_test_accuracies.append(fold_test_accuracy)

    avg_train_accuracy = np.mean(fold_train_accuracies)
    avg_test_accuracy = np.mean(fold_test_accuracies)

    return avg_train_accuracy, avg_test_accuracy

def find_best_value(test_accuracies, values_range):
    best_index = np.argmax(test_accuracies)
    best_value = values_range[best_index]
    return best_index, best_value


# ### 5.2. Nearest Neighbors Hyperparameter Tuning Curve

# Iterating KNN models with different neighbor values (3 to 29) using 10-fold cross-validation.

# In[26]:


knn = KNeighborsClassifier()

n_neighbors_values = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

train_accuracies = []
test_accuracies = []

for n_neighbors in n_neighbors_values:
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_index, best_n_neighbors = find_best_value(test_accuracies, n_neighbors_values)

plt.plot(n_neighbors_values, train_accuracies, label='Average Training Accuracy')
plt.plot(n_neighbors_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_n_neighbors, test_accuracies[best_index], color='red', marker='x', label=f'Best Test Accuracy (n_neighbors={best_n_neighbors})')
plt.xlabel('Number of Neighbors (n_neighbors)')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different n_neighbors Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best n_neighbors: {best_n_neighbors}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_index]}")


# ### 5.3. Weights Methods Hyperparameter Tuning Curve

# Iterating KNN models with different distance weights methods (uniform and distance) using 10-fold cross-validation.

# In[28]:


weights_values = ['uniform', 'distance']

train_accuracies = []
test_accuracies = []

for weights in weights_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=weights)  # Assuming 5 neighbors
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_weights_index, best_weights = find_best_value(test_accuracies, weights_values)

plt.plot(weights_values, train_accuracies, label='Average Training Accuracy')
plt.plot(weights_values, test_accuracies, label='Average Test Accuracy')
```

```
plt.scatter(best_weights, test_accuracies[best_weights_index], color='red', marker='x', label=f'Best Test Accuracy (weights={best_weights})')
plt.xlabel('Weights')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Weights Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best weights: {best_weights}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_weights_index]}")


# ### 5.4. Minkowski distance p values Hyperparameter Tuning Curve

# Iterating KNN models with different p values for minkowski distance (0.5 to 4.0) using 10-fold cross-validation.

# In[29]:


p_values = [i/2 for i in range(2, 9)]

train_accuracies = []
test_accuracies = []

for p in p_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=p, metric='minkowski')
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_p_index, best_p = find_best_value(test_accuracies, p_values)

plt.plot(p_values, train_accuracies, label='Average Training Accuracy')
plt.plot(p_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_p, test_accuracies[best_p_index], color='red', marker='x', label=f'Best Test Accuracy (p={best_p})')
plt.xlabel('p')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different p Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best p value: {best_p}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_p_index]}")


# ### 5.5. Spatial Algorithms Hyperparameter Tuning Curve

# Iterating KNN models with different spacial algorithm (auto, ball_tree, kd_tree, brute) using 10-fold cross-validation.

# In[30]:


algorithm_values = ['auto', 'ball_tree', 'kd_tree', 'brute']

train_accuracies = []
test_accuracies = []

for algorithm in algorithm_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=best_p, algorithm=algorithm)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_algorithm_index, best_algorithm = find_best_value(test_accuracies, algorithm_values)

plt.plot(algorithm_values, train_accuracies, label='Average Training Accuracy')
plt.plot(algorithm_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_algorithm, test_accuracies[best_algorithm_index], color='red', marker='x', label=f'Best Test Accuracy (algorithm={best_algorithm})')
plt.xlabel('Algorithm')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Algorithm Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best algorithm: {best_algorithm}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_algorithm_index]}")


# ### 5.6. Leaf Size Values Hyperparameter Tuning Curve

# Iterating KNN models with different leaf size values (10 to 40) using 10-fold cross-validation.

# In[31]:


leaf_size_values = [10, 20, 30, 40]

train_accuracies = []
test_accuracies = []

for leaf_size in leaf_size_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=best_p, algorithm=best_algorithm, leaf_size=leaf_size)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_leaf_size_index, best_leaf_size = find_best_value(test_accuracies, leaf_size_values)

plt.plot(leaf_size_values, train_accuracies, label='Average Training Accuracy')
plt.plot(leaf_size_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_leaf_size, test_accuracies[best_leaf_size_index], color='red', marker='x', label=f'Best Test Accuracy (leaf_size={best_leaf_size})')
plt.xlabel('Leaf Size')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Leaf Size Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best leaf_size: {best_leaf_size}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_leaf_size_index]}")


# ### 5.7. Distance Metric Hyperparameter Tuning Curve

# Iterating KNN models with different distance metric (euclidean, manhattan, minkowski with best_p value) using 10-fold cross-validation.

# In[32]:


metric_values = ['euclidean', 'manhattan', 'minkowski']

train_accuracies = []
```

```
test_accuracies = []

kf = KFold(n_splits=10, shuffle=True, random_state=42)

for metric in metric_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=best_p, algorithm=best_algorithm, leaf_size=best_leaf_size, metric=metric)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_metric_index = np.argmax(test_accuracies)

best_metric = metric_values[best_metric_index]

best_metric_index, best_metric = find_best_value(test_accuracies, metric_values)

plt.plot(metric_values, train_accuracies, label='Average Training Accuracy')
plt.plot(metric_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_metric, test_accuracies[best_metric_index], color='red', marker='x', label=f'Best Test Accuracy (metric={best_metric})')
plt.xlabel('Metric')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Metric Values with 10-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best metric: {best_metric}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_metric_index]}")


# ## 6. PCA Performance Trade-off

# Assessing PCA performance with varying number of components in a KNN model. It evaluates accuracy and records execution time, plotting results to evaluate trade-offs between accuracy

# In[50]:


import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score

n_components_values = list(range(1, 15))

accuracy_scores = []
execution_times = []

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

for n_components in n_components_values:
    pca = PCA(n_components=n_components)
    knn_pca = KNeighborsClassifier(
    algorithm=best_hyperparameters[0],
    leaf_size=best_hyperparameters[1],
    metric=best_hyperparameters[2],
    n_neighbors=best_hyperparameters[3],
    p=best_hyperparameters[4],
    weights=best_hyperparameters[5])
    pipeline = Pipeline([
        ('scaler', StandardScaler()),
        ('pca', pca),
        ('knn', knn_pca)
    ])

    start_time = time.time()

    scores = cross_val_score(pipeline, X_train, y_train, cv=kf, scoring='accuracy')

    accuracy_scores.append(np.mean(scores))
    execution_times.append(time.time() - start_time)

fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.set_xlabel('n_components')
ax1.set_ylabel('Accuracy', color=color)
ax1.plot(n_components_values, accuracy_scores, color=color, marker='o')
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()
color = 'tab:blue'
ax2.set_ylabel('Execution Time (seconds)', color=color)
ax2.plot(n_components_values, execution_times, color=color, marker='x')
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()
plt.title('PCA Performance for Different n_components Values')
plt.show()
```

4) Neural Network: Steps to run the code:

1) Run the file `main.py` in the directory UCI/Neural_Network
2) Run the file `plot.py` in the directory UCI/Neural_Network

Script ftns.py in the directory UCI/Neural_Network

```python
import ftns as ftns
from ucimlrepo import fetch_ucirepo
import torch as t
import torch
import torch.cuda as cuda
import torch.optim as optimizer
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transf
from torchsummary import summary
from torch.utils.data import DataLoader
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# Chatgpt prompt: Replace strings in a dataframe with one-hot labels.
def encode_strings_inplace(df):
    numeric_columns = df.select_dtypes(exclude=['object']).columns
    string_columns = df.select_dtypes(include=['object']).columns
    print("Numeric columns:", numeric_columns.tolist())
    print("String columns:", string_columns.tolist())
    for column in string_columns:
        unique_values = df[column].unique()
        encoding = {value: index + 1 for index, value in enumerate(unique_values)}
        df[column] = df[column].map(encoding)
        print(f"Unique values encoded for '{column}':", encoding)
    return df




def loading_dataset(shuffle):
 adult = fetch_ucirepo(id=2)
 X = adult.data.features
 y = adult.data.targets
 df_concatenated = pd.concat([X, y], axis=1)
 df_concatenated = df_concatenated.dropna()
 df_concatenated = df_concatenated[~df_concatenated.applymap(lambda x: x == '?').any(axis=1)]
 df_concatenated['income'] = df_concatenated['income'].map({'<=50K': 0, '<=50K.': 0, '>50K': 1 , '>50K.': 1})
 df_concatenated.to_csv('full_df_prelimineary.csv', index=False)
 df_concatenated = encode_strings_inplace(df_concatenated)
 df_concatenated.to_csv('full_df.csv', index=False)
 if shuffle == True:
  df_concatenated = df_concatenated.sample(frac=1).reset_index(drop=True)
 return df_concatenated


# Chatgpt prompt: How to split data into train, validation, and test sets by indices
def split_samples_with_indices_v2(total_samples, train_ratio=0.7, validation_ratio=0.2, test_ratio=0.1):
    train_samples = int(total_samples * train_ratio)
    validation_samples = int(total_samples * validation_ratio)
    test_samples = total_samples - train_samples - validation_samples
    train_start, train_end = 0, train_samples
    validation_start, validation_end = train_end, train_end + validation_samples
    test_start, test_end = validation_end, total_samples
    return train_samples, (train_start, train_end), validation_samples, (validation_start, validation_end), test_samples, (test_start, test_end)
```

Script plot.py in the directory UCI/Neural_Network

```python
import numpy as np
import matplotlib.pyplot as plt


def plot_array(arr1, label1, arr2, label2, title, xlabel, ylabel, fig_name):
    index = np.arange(1, len(arr1) + 1)
    plt.figure(figsize=(10, 6))
    plt.plot(index, arr1, '-o', color = 'orange' ,label=label1)
    plt.plot(index, arr2, '-o' ,label=label2)
    plt.legend()
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.grid(True)
    plt.savefig(fig_name)
    plt.show()




# train_loss = np.load('train_loss_arr.npy')
# val_loss = np.load('val_loss_arr.npy')

train_acc = np.load('train_acc_arr.npy')
val_acc = np.load('val_acc_arr.npy')

# plot_array(train_loss, 'Training Loss', val_loss, 'Validation Loss', 'Training and Validation loss vs Epochs', 'Epochs', 'Cross Entropy Loss', 'loss_vs_epochs_fashionmnist.png')
plot_array(train_acc, 'Training Accuracy', val_acc, 'Validation Accuracy', 'Training and Validation Accuracy vs Epochs', 'Epochs', 'Accuracy', 'accuracy_vs_epochs_UCI.png')
```

Script data_feed.py in the directory UCI/Neural_Network

```
import os
import random
import pandas as pd
import torch
import numpy as np
import random
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, utils
import ast



############## Create data sample list #################
def create_samples(X, samples_range):
    data_samples = []
    for idx, row in X[samples_range[0]:samples_range[1]].iterrows():
        data = list(row.values)
        data_samples.append(data)
    return data_samples



############################################################



############################################################
class DataFeed(Dataset):

    def __init__(self, X, samples_range):
        self.samples_range = samples_range
        self.samples = create_samples(X, samples_range)


    def __len__(self):
        return len( self.samples )


    def __getitem__(self, idx):
        sample = self.samples[idx]
        input_features = sample[:-1]
        output = sample[-1]
        input_arr = np.asarray(input_features)
        output_arr = np.asarray(output)


        return (input_arr, output_arr)
```

Script main.py

```
import ftns as ftns
from ucimlrepo import fetch_ucirepo
import os

import torch as t
import torch
import torch.cuda as cuda
import torch.optim as optimizer
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transf
from torchsummary import summary

from data_feed import DataFeed
from torch.utils.data import DataLoader

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
from sklearn.metrics import confusion_matrix
import seaborn as sns

def main():

    X = ftns.loading_dataset(shuffle=True)
    size_input = 14
    node = 175
    size_output = 2
    batch_size = 32
    val_batch_size = 1
    test_batch_size = 1
    lr = 0.01
    decay = 1e-4
    num_epochs = 25
    total_samples = X.shape[0]
    print('total_samples', total_samples)
    train_samples, train_indices, validation_samples, validation_indices, test_samples, test_indices = ftns.split_samples_with_indices_v2(
        total_samples)
    print()
    print('train_samples', train_samples)
    print('train_indices', train_indices)
    print('validation_samples', validation_samples)
    print('validation_indices', validation_indices)
    print('test_samples', test_samples)
    print('test_indices', test_indices)

    train_samples, train_indices, validation_samples, validation_indices, test_samples, test_indices
    print()


    train_loader = DataLoader(DataFeed(X, samples_range = train_indices),
                              batch_size=batch_size,
                              shuffle=False)

    val_loader = DataLoader(DataFeed(X, samples_range = validation_indices),
                            batch_size=val_batch_size,
                            shuffle=False)

    test_loader = DataLoader(DataFeed(X, samples_range = test_indices),
                             batch_size=test_batch_size,
                             shuffle=False)


    print('len(train_loader)', len(train_loader))
    print('len(val_loader)', len(val_loader))
    print('len(test_loader)', len(test_loader))
    print()

    class NN_pred(nn.Module):
        def __init__(self, num_features, num_output, node):
            super(NN_pred, self).__init__()

            self.layer_1 = nn.Linear(num_features, node)
            self.batchn1 = nn.BatchNorm1d(node)
            self.layer_2 = nn.Linear(node, node)
            self.batchn2 = nn.BatchNorm1d(node)
            self.layer_out = nn.Linear(node, num_output)
            self.relu = nn.ReLU()
            self.dropout = nn.Dropout(0.5)

        def forward(self, inputs):
            x = self.relu(self.batchn1(self.layer_1(inputs)))
            x = self.dropout(x)
            x = self.relu(self.batchn2(self.layer_2(x)))
            x = self.dropout(x)
            x = self.layer_out(x)
            return x

    device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
```

```
mod1 = NN_pred(size_input, size_output, node)
train_loss_lst = []
val_loss_lst = []
normalized_train_loss_lst = []
normalized_val_loss_lst = []
train_acc_lst = []
val_acc_lst = []


with cuda.device(1):
    top_1 = np.zeros((1, 1))
    acc_loss = 0
    net = mod1.cuda()
    print(summary(mod1, (size_input,)))
    layers = list(net.children())
    cost = nn.CrossEntropyLoss()
    opt = optimizer.Adam(net.parameters(), lr=lr, weight_decay=decay)

    count = 0
    current_top_acc = []
    best_accuracy = 0

    for epoch in range(num_epochs):
        train_loss = 0
        val_loss = 0
        train_correct = 0
        val_correct = 0
        train_total = 0
        val_total = 0
        print('Epoch No. ' + str(epoch + 1))
        skipped_batches = 0
        for jk, (inp1, outp1) in enumerate(train_loader):
            net.train()
            data = inp1.type(torch.Tensor)
            label = outp1.type(torch.LongTensor)
            x = data.cuda()
            opt.zero_grad()
            label = label.cuda()
            out = net.forward(x)
            loss = cost(out, label)
            train_loss += loss.item()
            loss.backward()
            opt.step()
            count += 1
            _, predicted = torch.max(out, 1)
            train_total += label.size(0)
            train_correct += (predicted == label).sum().item()

        normalized_train_loss = train_loss / len(train_loader)
        normalized_train_loss_lst.append(normalized_train_loss)




        print('Starting validation')
        ave_top_acc = 0
        toppred_out = []
        total_count = 0
        groundtruth = []


        for val_count, (inp1, outp1) in enumerate(val_loader):
            net.eval()
            data = inp1.type(torch.Tensor)
            x = data.cuda()
            labels = outp1.type(torch.LongTensor)
            opt.zero_grad()
            labels = labels.cuda()
            groundtruth.append(labels.detach().cpu().numpy()[0].tolist())
            total_count += labels.size(0)
            out = net.forward(x)
            val_loss += cost(out, labels).item()
            _, top_1_pred = t.max(out, dim=1)
            toppred_out.append(top_1_pred.detach().cpu().numpy()[0].tolist())
            sorted_out = t.argsort(out, dim=1, descending=True)

            reshaped_labels = labels.reshape((labels.shape[0], 1))

            bt_acc = t.sum(top_1_pred == labels, dtype=t.float32)

            ave_top_acc += bt_acc.item()

        normalized_val_loss = val_loss / len(val_loader)
        normalized_val_loss_lst.append(normalized_val_loss)


        print("total test examples are", total_count)
        print("total train examples are", train_total)
        current_top_acc.append(ave_top_acc / total_count)  # (batch_size * (count_2 + 1)) )
```

```
            train_acc_lst.append(train_correct/train_total)
            np.save('train_acc_arr.npy', np.array(train_acc_lst))
            val_acc_lst.append(ave_top_acc / total_count)
            np.save('val_acc_arr.npy', np.array(val_acc_lst))
            print('Average Top-1 accuracy {}'.format(current_top_acc[-1]))
            cur_accuracy = current_top_acc[-1]
            np.save('normalized_train_loss_arr.npy', np.array(normalized_train_loss_lst))  # Save normalized training loss
            np.save('normalized_val_loss_arr.npy', np.array(normalized_val_loss_lst))      # Save normalized validation loss



        all_labels = []
        all_predictions = []
        for val_count, (inp1, outp1) in enumerate(test_loader):
            net.eval()
            data = inp1.type(torch.Tensor)
            x = data.cuda()
            labels = outp1.type(torch.LongTensor)
            opt.zero_grad()
            labels = labels.cuda()
            groundtruth.append(labels.detach().cpu().numpy()[0].tolist())
            total_count += labels.size(0)
            out = net.forward(x)
            _, top_1_pred = t.max(out, dim=1)
            toppred_out.append(top_1_pred.detach().cpu().numpy()[0].tolist())
            sorted_out = t.argsort(out, dim=1, descending=True)

            reshaped_labels = labels.reshape((labels.shape[0], 1))

            bt_acc = t.sum(top_1_pred == labels, dtype=t.float32)
            ave_top_acc += bt_acc.item()
            #Chatgpt prompt: How to plot a confusion matrix given labels and groundtruths from a for loop.
            all_labels.extend(labels.cpu().numpy())
            all_predictions.extend(top_1_pred.cpu().numpy())


        testing_accuracy = ave_top_acc / total_count
        print('total test examples are', total_count)
        print('testing_accuracy', testing_accuracy)
        np.save('final testing accuracy', testing_accuracy)


        cm = confusion_matrix(all_labels, all_predictions)
        plt.figure(figsize=(10, 7))
        annot_kws = {"fontsize": 25}
        ax = sns.heatmap(cm, annot=True, fmt='g', cbar=False, annot_kws=annot_kws)
        plt.title('Confusion Matrix')
        plt.xlabel('Predicted Labels')
        plt.ylabel('True Labels')
        class_labels = ['<=50K', '>50K']
        ax.set_xticklabels(class_labels)
        ax.set_yticklabels(class_labels)
        plt.savefig('Confusion Matrix.png')
        plt.show()









main()
```

C. FashionMNIST Dataset

1) Logistic Regression: Steps to run the code:

1) Run the file `SSHAH_TermProject_Fashion_LogisticRegression_GD_Final.ipynb` in the directory FashionMNIST/Logistic_Regression
2) Run the file `SSHAH_TermProject_Fashion_LogisticRegression_SGD_Final.ipynb` in the directory FashionMNIST/Logistic_Regression

```
# -*- coding: utf-8 -*-
"""SSHAH_TermProject_Fashion_LogisticRegression_GD_Final.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1JBWTFPyric5-YntsVsCitGlwv8UFKPVL

## Importing Data and Libraries
"""

import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score, recall_score, f1_score

import matplotlib.pyplot as plt
import seaborn as sns
import torch
import torch.nn as nn
import torchvision
import tensorflow as tf

import copy
import time

# Loading the FashionMNIST data
train_set = torchvision.datasets.FashionMNIST("./data", download=True, train=True)
test_set = torchvision.datasets.FashionMNIST("./data", download=True, train=False)
X_train = train_set.data.numpy()
y_train = train_set.targets.numpy()
X_test = test_set.data.numpy()
y_test = test_set.targets.numpy()

"""## Pre-processing Data: Cleaning, Equalizing, and LabelEncoding"""

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

meanValue = np.mean(X_train)
stdValue = np.std(X_train)

X_train = (X_train - meanValue) / stdValue
X_test = (X_test - meanValue) / stdValue

print(X_train.shape)
print(X_test.shape)

pca = PCA()
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

varianceRatio = np.cumsum(pca.explained_variance_ratio_)
threshold = 0.90

principalVectors = np.argmax(varianceRatio >= threshold) + 1
X_train_reduced = X_train_pca[:, :principalVectors]
X_test_reduced = X_test_pca[:, :principalVectors]

plt.plot(varianceRatio)
plt.xlabel('q - # of Principal Vectors')
```

```
plt.ylabel('Cumulative Explained Variance Ratio')
plt.title('Cumulative Explained Variance Ratio vs. # of Principal Vectors')
plt.grid(True)
plt.show()
print(f'# of principal vectors for {threshold*100}% of the variance: {principalVectors}')

"""## Model Construction and Training"""

class MultinomialLogisticRegression(nn.Module):
    def __init__(self, num_features, num_classes, l2=0.0):
        super(MultinomialLogisticRegression, self).__init__()
        self.layer1 = nn.Linear(num_features, num_classes)
        self.l2 = l2

    def forward(self, x):
        out = self.layer1(x)
        return out

def resetWeights(model):
    for layer in model.children():
        if isinstance(layer, nn.Linear):
            layer.reset_parameters()

def trainModel(model, epochs, lossFn, X_train, X_test, y_train, y_test, lr=0.01, batchSize=32,
decisionThreshold=0.5):
    trainAcc, testAcc = np.zeros(epochs), np.zeros(epochs)
    trainErr, testErr = np.zeros(epochs), np.zeros(epochs)
    trainLoss, testLoss = np.zeros(epochs), np.zeros(epochs)
    batchCount = 0

    resetWeights(model)
    train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
    train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batchSize,
shuffle=True)
    for epoch in range(epochs):
        for batch_X, batch_y in train_loader:
          batchCount += 1
          output = model(batch_X)
          loss = lossFn(output.squeeze(), batch_y)
          loss.backward()

          with torch.no_grad():
              for param in model.parameters():
                  param.data -= lr * param.grad

              model.zero_grad()

          with torch.no_grad():
              trainLoss[epoch] += loss.item()
              _, trainPredictions = torch.max(output, 1)
              trainError = torch.mean((trainPredictions != batch_y).float())

              output = model(X_test)
              loss = lossFn(output.squeeze(), y_test)

              testLoss[epoch] += loss.item()
              _, testPredictions = torch.max(output, 1)
              testError = torch.mean((testPredictions != y_test).float())

              trainErr[epoch] += trainError.item()
              testErr[epoch] += testError.item()
              trainAcc[epoch] += 1 - trainError.item()
              testAcc[epoch] += 1 - testError.item()

    trainAcc = trainAcc/(batchCount/epochs)
    testAcc = testAcc/(batchCount/epochs)
    trainErr = trainErr/(batchCount/epochs)
```

```
    testErr = testErr/(batchCount/epochs)
    trainLoss = trainLoss/(batchCount/epochs)
    testLoss = testLoss/(batchCount/epochs)

    return trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss


X_train_tensor = torch.tensor(X_train_reduced).float()
y_train_tensor = torch.tensor(y_train).long()
X_test_tensor = torch.tensor(X_test_reduced).float()
y_test_tensor = torch.tensor(y_test).long()

"""## Grid Search Analysis"""

epochs_list = [50, 100, 250]
learning_rates = [1, 0.5, 0.05]
batchSizes = [4096]
lambdas = [0, 0.001, 0.1]
trainedModels = []

num_features = X_train_tensor.shape[1]
num_classes = len(set(y_train))
lossFn = nn.CrossEntropyLoss()

for lambda_ in lambdas:
  model = MultinomialLogisticRegression(num_features, num_classes, l2=lambda_)
  for batchSize in batchSizes:
    for lr in learning_rates:
        for epochs in epochs_list:
          startTime = time.time()
          trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModel(
              model,
              epochs,
              lossFn,
              X_train_tensor,
              X_test_tensor,
              y_train_tensor,
              y_test_tensor,
              lr=lr,
              batchSize=batchSize
          )
          trainingTime = time.time() - startTime
          trainedModels.append({ 'model': copy.deepcopy(model),
                                'hyperparameters': {'epochs': epochs, 'lr': lr, 'batchSize':
batchSize, 'lambda': lambda_ },
                                'trainingStats': { 'testAcc': testAcc[-1], 'trainAcc':
trainAcc[-1],
                                                   'testError': testErr[-1], 'trainError':
trainErr[-1],
                                                   'testLoss': testLoss[-1], 'trainLoss':
trainLoss[-1] },
                                'trainingTime': trainingTime
                                })
          print(f'Completed epochs, lr, batchSize, lambda, test error: {epochs}, {lr},
{batchSize}, {lambda_}, --- {testAcc[-1]}')

"""# Finding the best model that maximizes accuracy

"""

globalOptimalModel = None
globalOptimalLoss = None
globalOptimalAcc = None

for model_ in trainedModels:
    optimalAcc = model_['trainingStats']['testAcc']
    optimalLoss = model_['trainingStats']['testLoss']
```

```
    if (globalOptimalLoss == None):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
    elif (optimalAcc > globalOptimalAcc):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
    elif (optimalAcc == globalOptimalAcc and optimalLoss < globalOptimalLoss):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_

print(globalOptimalModel)

gLr = globalOptimalModel['hyperparameters']['lr']
gEpochs = globalOptimalModel['hyperparameters']['epochs']
gBatchSize = globalOptimalModel['hyperparameters']['batchSize']
gLambda = globalOptimalModel['hyperparameters']['lambda']

print(f'Global Optimal Model Hyperparameters: lr={gLr}, epochs={gEpochs}, batchSize=
{gBatchSize}, lamda={gLambda}')
print(f'Global Optimal Acc: {globalOptimalAcc}, Global Optimal Loss: {globalOptimalLoss}')

gModel = globalOptimalModel['model']
with torch.no_grad(): output = gModel(X_test_tensor)
_, testPredictions = torch.max(output, 1)
y_true = np.array(y_test_tensor)
y_pred = np.array(testPredictions)
cm = confusion_matrix(y_true, y_pred)
precision = precision_score(y_true, y_pred, average='weighted')
recall = recall_score(y_true, y_pred, average='weighted')
f1 = f1_score(y_true, y_pred, average='weighted')
labels = class_labels = ['T-
shirt/top','Trouser','Pullover','Dress','Coat','Sandal','Shirt','Sneaker','Bag','Ankle boot']

# Display the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix for Multinomial Logistic Regression')
plt.show()

print(f'Precision: {precision}, Recall: {recall}, F1: {f1}')

"""### Reproducability Test & Visualization"""

lr = globalOptimalModel['hyperparameters']['lr']
epochs = globalOptimalModel['hyperparameters']['epochs']
batchSize = globalOptimalModel['hyperparameters']['batchSize']
lambda_ = globalOptimalModel['hyperparameters']['lambda']

num_features = X_train_tensor.shape[1]
num_classes = len(set(y_train))
lossFn = nn.CrossEntropyLoss()
model = MultinomialLogisticRegression(num_features, num_classes, l2=lambda_)
lossFn = nn.CrossEntropyLoss()

trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModel(
    model,
    epochs,
    lossFn,
    X_train_tensor,
    X_test_tensor,
    y_train_tensor,
```

```
    y_test_tensor,
    lr=lr,
    batchSize=batchSize
)

model.eval()

plt.plot(trainAcc, label='train accuracy')
plt.plot(testAcc, label='test accuracy')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainErr, label='train error')
plt.plot(testErr, label='test error')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainLoss, label='train loss')
plt.plot(testLoss, label='test loss')
plt.grid(True)
plt.legend()

plt.show()

print(f'Global Optimal Model Hyperparameters: lr={lr}, epochs={epochs}, batchSize={batchSize},
lamda={lambda_}')
print(f'Train Accuracy: {trainAcc[-1]}, Train Error: {trainErr[-1]}, Train Loss: {trainLoss[-
1]}')
print(f'Test Accuracy: {testAcc[-1]}, Test Error: {testErr[-1]}, Test Loss: {testLoss[-1]}')

"""## Time Complexity Analysis"""

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

lambdaValue = 0.001
filteredModels = [modelInfo for modelInfo in trainedModels if modelInfo['hyperparameters']
['lambda'] == lambdaValue]

# Extract hyperparameters and training time
epochsList = [modelInfo['hyperparameters']['epochs'] for modelInfo in filteredModels]
learningRates = [modelInfo['hyperparameters']['lr'] for modelInfo in filteredModels]
batchSizes = [modelInfo['hyperparameters']['batchSize'] for modelInfo in filteredModels]
trainingTimes = [modelInfo['trainingTime'] for modelInfo in filteredModels]

# Create 3D plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot with color based on training time
sc = ax.scatter(epochsList, learningRates, batchSizes, c=trainingTimes, cmap='viridis',
marker='o', s=100)

# Set labels and title
ax.set_xlabel('Epochs')
ax.set_ylabel('Learning Rate')
ax.set_zlabel('Batch Size')
ax.set_title('Training Time Variation with Hyperparameters')

# Add colorbar
cbar = plt.colorbar(sc)
cbar.set_label('Training Time (seconds)')

plt.show()
```

```
# -*- coding: utf-8 -*-
"""SSHAH_TermProject_Fashion_LogisticRegression_SGD_Final.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1k8MXho3gf7V_jqmzcRWTGExpnb8uH9za

# Importing Data and Libraries
"""

import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score, recall_score, f1_score

import matplotlib.pyplot as plt
import seaborn as sns
import torch
import torch.nn as nn
import torchvision
import tensorflow as tf

import copy
import time

# Loading the FashionMNIST data
train_set = torchvision.datasets.FashionMNIST("./data", download=True, train=True)
test_set = torchvision.datasets.FashionMNIST("./data", download=True, train=False)
X_train = train_set.data.numpy()
y_train = train_set.targets.numpy()
X_test = test_set.data.numpy()
y_test = test_set.targets.numpy()

"""# Pre-processing Data: Cleaning, Equalizing, and LabelEncoding"""

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

meanValue = np.mean(X_train)
stdValue = np.std(X_train)

X_train = (X_train - meanValue) / stdValue
X_test = (X_test - meanValue) / stdValue

print(X_train.shape)
print(X_test.shape)

pca = PCA()
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

varianceRatio = np.cumsum(pca.explained_variance_ratio_)
threshold = 0.90

principalVectors = np.argmax(varianceRatio >= threshold) + 1
X_train_reduced = X_train_pca[:, :principalVectors]
X_test_reduced = X_test_pca[:, :principalVectors]

plt.plot(varianceRatio)
plt.xlabel('q - # of Principal Vectors')
```

```
plt.ylabel('Cumulative Explained Variance Ratio')
plt.title('Cumulative Explained Variance Ratio vs. # of Principal Vectors')
plt.grid(True)
plt.show()
print(f'# of principal vectors for {threshold*100}% of the variance: {principalVectors}')

"""# Model Construction and Training"""

class MultinomialLogisticRegression(nn.Module):
    def __init__(self, num_features, num_classes, l2=0.0):
        super(MultinomialLogisticRegression, self).__init__()
        self.layer1 = nn.Linear(num_features, num_classes)
        self.l2 = l2

    def forward(self, x):
        out = self.layer1(x)
        return out

def resetWeights(model):
    for layer in model.children():
        if isinstance(layer, nn.Linear):
            layer.reset_parameters()

def trainModel(model, epochs, lossFn, optimizer, X_train, X_test, y_train, y_test,
decisionThreshold=0.5):
    trainAcc, testAcc = [], []
    trainErr, testErr = [], []
    trainLoss, testLoss = [], []

    resetWeights(model)
    for epoch in range(epochs):
        output = model(X_train)
        loss = lossFn(output.squeeze(), y_train)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        with torch.no_grad():
            trainLoss.append(loss.item())
            _, trainPredictions = torch.max(output, 1)
            trainError = torch.mean((trainPredictions != y_train_tensor).float())

            output = model(X_test)
            loss = lossFn(output.squeeze(), y_test)

            testLoss.append(loss.item())
            _, testPredictions = torch.max(output, 1)
            testError = torch.mean((testPredictions != y_test_tensor).float())

            trainErr.append(trainError.item())
            testErr.append(testError.item())
            trainAcc.append(1 - trainError.item())
            testAcc.append(1 - testError.item())

    return trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss


X_train_tensor = torch.tensor(X_train_reduced).float()
y_train_tensor = torch.tensor(y_train).long()
X_test_tensor = torch.tensor(X_test_reduced).float()
y_test_tensor = torch.tensor(y_test).long()

"""# Grid Search Analysis"""

epochs_list = [50, 100, 500, 1000]
learning_rates = [1, 0.1, 0.01, 0.001]
momentums = [0, 0.5, 0.9, 0.99]
```

```
lambdas = [0, 0.001, 0.01, 0.1, 1]
trainedModels = []

num_features = X_train_tensor.shape[1]
num_classes = len(set(y_train))
lossFn = nn.CrossEntropyLoss()

for lambda_ in lambdas:
  model = MultinomialLogisticRegression(num_features, num_classes, l2=lambda_)
  for momentum in momentums:
    for lr in learning_rates:
        optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum,
weight_decay=model.l2)
        for epochs in epochs_list:
          startTime = time.time()
          trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModel(
              model,
              epochs,
              lossFn,
              optimizer,
              X_train_tensor,
              X_test_tensor,
              y_train_tensor,
              y_test_tensor
          )
          trainingTime = time.time() - startTime
          trainedModels.append({ 'model': copy.deepcopy(model),
                                 'hyperparameters': {'epochs': epochs, 'lr': lr, 'momentum':
momentum, 'lambda': lambda_ },
                                 'trainingStats': { 'testAcc': testAcc[-1], 'trainAcc':
trainAcc[-1],
                                                    'testError': testErr[-1], 'trainError':
trainErr[-1],
                                                    'testLoss': testLoss[-1], 'trainLoss':
trainLoss[-1] },
                                 'trainingTime': trainingTime
                                 })
          print(f'Completed epochs, lr, momentum, lambda, test error: {epochs}, {lr},
{momentum}, {lambda_}, --- {testAcc[-1]}')

"""# Finding the best model that maximizes accuracy


"""

globalOptimalModel = None
globalOptimalLoss = None
globalOptimalAcc = None

for model_ in trainedModels:
    optimalAcc = model_['trainingStats']['testAcc']
    optimalLoss = model_['trainingStats']['testLoss']

    if (globalOptimalLoss == None):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
    elif (optimalAcc > globalOptimalAcc):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_
    elif (optimalAcc == globalOptimalAcc and optimalLoss < globalOptimalLoss):
      globalOptimalLoss = optimalLoss
      globalOptimalAcc = optimalAcc
      globalOptimalModel = model_

print(globalOptimalModel)
```

```
"""###33"""

gLr = globalOptimalModel['hyperparameters']['lr']
gEpochs = globalOptimalModel['hyperparameters']['epochs']
gMomentum = globalOptimalModel['hyperparameters']['momentum']
gLambda = globalOptimalModel['hyperparameters']['lambda']

print(f'Global Optimal Model Hyperparameters: lr={gLr}, epochs={gEpochs}, momentum={gMomentum},
lamda={gLambda}')
print(f'Global Optimal Acc: {globalOptimalAcc}, Global Optimal Loss: {globalOptimalLoss}')

gModel = globalOptimalModel['model']
with torch.no_grad(): output = gModel(X_test_tensor)
_, testPredictions = torch.max(output, 1)
y_true = np.array(y_test_tensor)
y_pred = np.array(testPredictions)
cm = confusion_matrix(y_true, y_pred)
precision = precision_score(y_true, y_pred, average='weighted')
recall = recall_score(y_true, y_pred, average='weighted')
f1 = f1_score(y_true, y_pred, average='weighted')
labels = class_labels = ['T-
shirt/top','Trouser','Pullover','Dress','Coat','Sandal','Shirt','Sneaker','Bag','Ankle boot']

# Display the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix for Multinomial Logistic Regression')
plt.show()

print(f'Precision: {precision}, Recall: {recall}, F1: {f1}')

"""## Reproducability Test & Visualizing Training Curves"""

lr = globalOptimalModel['hyperparameters']['lr']
epochs = globalOptimalModel['hyperparameters']['epochs']
momentum = globalOptimalModel['hyperparameters']['momentum']
lambda_ = globalOptimalModel['hyperparameters']['lambda']


num_features = X_train_tensor.shape[1]
num_classes = len(set(y_train))
model = MultinomialLogisticRegression(num_features, num_classes, l2=lambda_)
lossFn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum,
weight_decay=model.l2)

trainAcc, testAcc, trainErr, testErr, trainLoss, testLoss = trainModel(
    model,
    epochs,
    lossFn,
    optimizer,
    X_train_tensor,
    X_test_tensor,
    y_train_tensor,
    y_test_tensor
)

model.eval()

plt.plot(trainAcc, label='train accuracy')
plt.plot(testAcc, label='test accuracy')
plt.grid(True)
plt.legend()
```

```
plt.figure()
plt.plot(trainErr, label='train error')
plt.plot(testErr, label='test error')
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(trainLoss, label='train loss')
plt.plot(testLoss, label='test loss')
plt.grid(True)
plt.legend()

plt.show()

print(f'Global Optimal Model Hyperparameters: lr={lr}, epochs={epochs}, momentum={momentum},
lamda={lambda_}')
print(f'Train Accuracy: {trainAcc[-1]}, Train Error: {trainErr[-1]}, Train Loss: {trainLoss[-
1]}')
print(f'Test Accuracy: {testAcc[-1]}, Test Error: {testErr[-1]}, Test Loss: {testLoss[-1]}')

"""# Time Complexity Analysis"""

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

lambdaValue = 0.001
filteredModels = [modelInfo for modelInfo in trainedModels if modelInfo['hyperparameters']
['lambda'] == lambdaValue]

epochsList = [modelInfo['hyperparameters']['epochs'] for modelInfo in filteredModels]
learningRates = [modelInfo['hyperparameters']['lr'] for modelInfo in filteredModels]
momentums = [modelInfo['hyperparameters']['momentum'] for modelInfo in filteredModels]
trainingTimes = [modelInfo['trainingTime'] for modelInfo in filteredModels]

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
sc = ax.scatter(epochsList, learningRates, momentums, c=trainingTimes, cmap='viridis',
marker='o', s=100)
ax.set_xlabel('Epochs')
ax.set_ylabel('Learning Rate')
ax.set_zlabel('Momentum')
ax.set_title('Training Time Variation with Hyperparameters')
cbar = plt.colorbar(sc)
cbar.set_label('Training Time (seconds)')

plt.show()
```

2) Support Vector Machine: Steps to run the code:

1) Run the file `Fashion_MNIST_SVM.ipynb` in the directory FashionMNIST/SVM. For running the file, the user just have to execute the "Run All Below" command, mentioned in the dropdown menu of the Cell tab in the notebook.

```
# -*- coding: utf-8 -*-
"""Fashion_MNIST_SVM.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1jfGIsyrt777oRl9nWARDG0leGir_6CPQ
"""

import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'
import numpy as np
import pandas as pd
import torch
import torchvision
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import learning_curve
from sklearn.decomposition import PCA
from scipy.stats import randint

# Loading the FashionMNIST data
train_set = torchvision.datasets.FashionMNIST("./data", download=True)
test_set = torchvision.datasets.FashionMNIST("./data", download=True, train=False)
X_train = train_set.data.numpy()
labels_train = train_set.targets.numpy()
X_test = test_set.data.numpy()
labels_test = test_set.targets.numpy()

X_train = X_train.reshape((X_train.shape[0], -1)) / 255.0
X_test = X_test.reshape((X_test.shape[0], -1)) / 255.0

y_train = labels_train
y_test = labels_test

# Standardizing the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Performing PCA before SVM

num_components = 20
pca = PCA(n_components=num_components)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)
explained_variance_ratio = pca.explained_variance_ratio_
print("Explained Variance Ratios:", explained_variance_ratio)

cumulative_variance = explained_variance_ratio.cumsum()

total_explained_variance_percentage = cumulative_variance[-1] * 100
print(f"Total Explained Variance Percentage: {total_explained_variance_percentage:.2f}%")

# Performing SVM without kernel

# Defining the hyperparameter
param_grid_linear = {
    'C': randint(1, 11)
}

svm_linear = SVC(kernel='linear')
```

```
random_search_linear = RandomizedSearchCV(estimator=svm_linear,
param_distributions=param_grid_linear,
                                          n_iter=5, cv=5, scoring='accuracy', verbose=2,
n_jobs=-1)
random_search_linear.fit(X_train_pca, y_train)

best_svm_linear = random_search_linear.best_estimator_

y_pred_linear = best_svm_linear.predict(X_test_pca)

# Evaluating the performance of the linear SVM
accuracy_linear = accuracy_score(y_test, y_pred_linear)
print("Best Linear SVM Accuracy:", accuracy_linear)
print("Classification Report (Best Linear SVM):")
print(classification_report(y_test, y_pred_linear))
best_C_linear = random_search_linear.best_params_['C']
print("Best C:", best_C_linear)

C_values = list(range(1, 11))
train_scores, test_scores = [], []

for C in C_values:
    best_svm_linear.C = C
    best_svm_linear.fit(X_train_pca, y_train)

    train_score = best_svm_linear.score(X_train_pca, y_train)
    train_scores.append(train_score)

    test_score = best_svm_linear.score(X_test_pca, y_test)
    test_scores.append(test_score)

# Plotting the training and test scores against the hyperparameter C
plt.figure(figsize=(10, 6))
plt.plot(C_values, train_scores, 'o-', label='Training Score')
plt.plot(C_values, test_scores, 'o-', label='Test Score')
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Test Scores vs C')
plt.show()

# Plotting the learning curve
def plot_learning_curve(estimator, X, y, cv, train_sizes=np.linspace(0.1, 1.0, 3)):
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, train_sizes=train_sizes, scoring='accuracy', n_jobs=-1
    )

    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_scores_mean, 'o-', color='r', label='Training Score')
    plt.plot(train_sizes, test_scores_mean, 'o-', color='g', label='Validation Score')

    plt.title('Learning Curve')
    plt.xlabel('Training Examples')
    plt.ylabel('Score')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()

plot_learning_curve(best_svm_linear, X_train_pca, y_train, cv=5)

# Performing SVM with Polynomial Kernel
```

```
# Defining the hyperparameters
param_dist_poly = {
    'C': randint(5, 8),
    'degree': np.random.choice([1, 2, 3, 5, 10], size=1)
}

svm_poly = SVC(kernel='poly')

random_search_poly = RandomizedSearchCV(estimator=svm_poly,
param_distributions=param_dist_poly,
                                        n_iter=5, cv=5, scoring='accuracy', verbose=2,
n_jobs=-1)
random_search_poly.fit(X_train_pca, y_train)

best_svm_poly = random_search_poly.best_estimator_

y_pred_poly = best_svm_poly.predict(X_test_pca)

# Evaluating the performance of Polynomial SVM
accuracy_poly = accuracy_score(y_test, y_pred_poly)
print("Best Polynomial SVM Accuracy:", accuracy_poly)
print("Classification Report (Best Polynomial SVM):")
print(classification_report(y_test, y_pred_poly))
best_degree = random_search_poly.best_params_['degree']
print("Best Degree:", best_degree)
best_C_poly = random_search_poly.best_params_['C']
print("Best C:", best_C_poly)


# Plotting the training and test scores against polynomial degree
degrees = [1, 2, 3, 5, 10]
train_scores, test_scores = [], []

for degree in degrees:
    best_svm_poly.degree = degree
    best_svm_poly.fit(X_train_pca, y_train)

    train_score = best_svm_poly.score(X_train_pca, y_train)
    train_scores.append(train_score)

    test_score = best_svm_poly.score(X_test_pca, y_test)
    test_scores.append(test_score)

plt.figure(figsize=(10, 6))
plt.plot(degrees, train_scores, 'o-', label='Training Score')
plt.plot(degrees, test_scores, 'o-', label='Test Score')
plt.xlabel('Degree of polynomial')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Test Scores vs Degree for Polynomial SVM')
plt.show()

# Plotting the learning curve
plot_learning_curve(best_svm_poly, X_train_pca, y_train, cv=5)

# Performing SVM with RBF Kernel

# Defining the hyperparameters
param_grid_rbf = {
    'C': randint(5, 8),
    'gamma': np.random.choice([0.1, 0.15, 0.2, 0.25, 0.3], size=1)
}

svm_rbf = SVC(kernel='rbf')

random_search_rbf = RandomizedSearchCV(estimator=svm_rbf, param_distributions=param_grid_rbf,
```

```
                                        n_iter=5, cv=5, scoring='accuracy', verbose=2,
n_jobs=-1)
random_search_rbf.fit(X_train_pca, y_train)

best_svm_rbf = random_search_rbf.best_estimator_

y_pred_rbf = best_svm_rbf.predict(X_test_pca)

# Evaluating the performance of RBF SVM
accuracy_rbf = accuracy_score(y_test, y_pred_rbf)
print("Best RBF SVM Accuracy:", accuracy_rbf)
print("Classification Report (Best RBF SVM):")
print(classification_report(y_test, y_pred_rbf))
best_gamma = random_search_rbf.best_params_['gamma']
print("Best Gamma:", best_gamma)
best_C_rbf = random_search_rbf.best_params_['C']
print("Best C:", best_C_rbf)

gammas = [0.1, 0.15, 0.2, 0.25, 0.3]
train_scores, test_scores = [], []

for gamma in gammas:
    best_svm_rbf.gamma = gamma
    best_svm_rbf.fit(X_train_pca, y_train)

    train_score = best_svm_rbf.score(X_train_pca, y_train)
    train_scores.append(train_score)

    test_score = best_svm_rbf.score(X_test_pca, y_test)
    test_scores.append(test_score)

# Plotting the training and test scores against gamma
plt.figure(figsize=(10, 6))
plt.plot(gammas, train_scores, 'o-', label='Training Score')
plt.plot(gammas, test_scores, 'o-', label='Test Score')
plt.xlabel('Gamma of RBF')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Test Scores vs Gamma for RBF SVM')
plt.show()

# Plotting the learning curve
def plot_learning_curve(estimator, X, y, cv, train_sizes=np.linspace(0.1, 1.0, 3)):
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, train_sizes=train_sizes, scoring='accuracy', n_jobs=-1
    )

    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_scores_mean, 'o-', color='r', label='Training Score')
    plt.plot(train_sizes, test_scores_mean, 'o-', color='g', label='Validation Score')

    plt.title('Learning Curve')
    plt.xlabel('Training Examples')
    plt.ylabel('Score')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()
# Plotting the learning curve
plot_learning_curve(best_svm_rbf, X_train_pca, y_train, cv=5)

# Plotting the confusion matrix for Polynomial SVM
```

```
from sklearn.metrics import confusion_matrix

best_svm_poly = SVC(kernel='poly', degree=5, C=7)

best_svm_poly.fit(X_train_pca, y_train)

y_pred_poly = best_svm_poly.predict(X_test_pca)

conf_matrix_poly = confusion_matrix(y_test, y_pred_poly)

plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix_poly, annot=True, fmt='d', cmap='Blues',
xticklabels=np.unique(y_test), yticklabels=np.unique(y_test))
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Polynomial SVM')
plt.show()
```

3) k-Nearest Neighbours:

1) Run the file `TMahmood_TermProject_Fashion_kNN.ipynb` in the directory FashionMNIST/kNN. For running the file, the user just have to execute the "Run All Below" command, mentioned in the dropdown menu of the Cell tab in the notebook.

```
#!/usr/bin/env python
# coding: utf-8

# ## 1. Importing Libraries

# In[1]:


from ucimlrepo import fetch_ucirepo
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
from sklearn.model_selection import KFold
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score


# ## 2. Importing Data

# In[2]:


import numpy as np
import torch, torchvision
import pandas as pd

train_set = torchvision.datasets.FashionMNIST("./data", download=True)
test_set = torchvision.datasets.FashionMNIST("./data", download=True, train=False)
X_train = train_set.data.numpy()
labels_train = train_set.targets.numpy()
X_test = test_set.data.numpy()
labels_test = test_set.targets.numpy()
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1]*X_train.shape[2]))
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1]*X_test.shape[2]))
X_train = X_train/255.0
X_test = X_test/255.0


# ## 3. Data Preprocessing

# ### 3.1. Scalarization of the Dataset

# Applying standardization to the feature matrix (X) using StandardScaler,. This ensures consistent scaling, aiding machine learning model performance.

# In[18]:


scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)


# ### 3.2. Principal Component Analysis

# Using Principal Component Analysis (PCA) on the scaled features, visualizing cumulative explained variance of principal components for dimensionality reduction.

# In[19]:


pca = PCA()
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

explained_variance_ratio = pca.explained_variance_ratio_

cumulative_variance = np.cumsum(explained_variance_ratio)
cutoff_index = np.argmax(cumulative_variance >= 0.90) + 1

plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, marker='o', linestyle='--')
plt.title('Cumulative Explained Variance')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Variance Explained')
plt.grid(True)
plt.show()


# Visualizing the explained variance ratio for each principal component using a bar plot and cumulative step plot.

# In[20]:


plt.bar(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio, alpha=0.75, align='center')
plt.step(range(1, len(explained_variance_ratio) + 1), np.cumsum(explained_variance_ratio), where='mid')
plt.title('Explained Variance Ratio for Each Principal Component')
plt.xlabel('Principal Component Number')
plt.ylabel('Explained Variance Ratio')
plt.savefig("Variance_Ratio")


# Creating a bar plot to visualize the absolute loadings of Principal Component 1 (PC1) in a PCA-transformed dataset, highlighting key features and their contributions to the variance

# In[21]:


merged_train_data = np.column_stack((X_train, labels_train))

columns = [f"pixel_{i}" for i in range(X_train.shape[1])] + ["label"]
merged_train_data = pd.DataFrame(data=merged_train_data, columns=columns)
train_data = merged_train_data.drop(columns=["label"])
feature_names = train_data.columns.tolist()

loadings_df = pd.DataFrame(pca.components_, columns=feature_names)

first_30_features = feature_names[:30]
```

```
loadings_df = loadings_df[first_30_features]

sorted_feature_names = loadings_df.abs().iloc[0, :].sort_values(ascending=False).index
sorted_loadings = loadings_df[sorted_feature_names]

plt.bar(sorted_feature_names, np.abs(sorted_loadings.iloc[0, :]))
plt.title('Absolute Loadings for PC1')
plt.xlabel('Features')
plt.ylabel('Absolute Loading')
plt.xticks(rotation=45, ha='right')
plt.show()Performing PCA to retain 90% of the variance in the scaled dataset.


# Performing PCA to retain 90% of the variance in the scaled dataset.

# In[23]:


cumulative_variance_ratio = np.cumsum(pca.explained_variance_ratio_)
n_components_90 = np.argmax(cumulative_variance_ratio >= 0.9) + 1
print(n_components_90)

pca = PCA(n_components=n_components_90)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)


# In[35]:


X_train = X_train_pca.copy()
X_test = X_test_pca.copy()
y_train = labels_train.copy()
y_test = labels_test.copy()


# ## 4. Model Training (Grid Search)

# ### 4.1. Defining the model and hyperparameters

# Defining the function to initiate kFold and knn, and assiging the hyperparameter range in the param_grid dictionary.

# In[40]:


knn = KNeighborsClassifier()
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11, 13, 15],
    'weights': ['uniform', 'distance'],
    'p': [i/2 for i in range(2, 9)],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'leaf_size': [10, 20, 30, 40],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}
kf = KFold(n_splits=3, shuffle=True, random_state=22)


# ### 4.2. Determing the best model based on validation accuracy

# Examining different hyperparameter combinations and their accuracies through a grid search. The bar plot displays the top 5 configurations with their accuracy scores, helping identi

# In[ ]:


from sklearn.model_selection import RandomizedSearchCV
grid_search = RandomizedSearchCV(knn, param_distributions=param_grid, n_iter=20, cv=kf, scoring='accuracy', random_state=22)

results = grid_search.fit(X_train, y_train)


# In[70]:


hyperparameters = []
accuracies = []

for params, mean_score, _ in zip(grid_search.cv_results_['params'], grid_search.cv_results_['mean_test_score'], grid_search.cv_results_['std_test_score']):
    params_tuple = tuple(params.values())

    hyperparameters.append(params_tuple)
    accuracies.append(mean_score)

results_df = pd.DataFrame({'Hyperparameters': hyperparameters, 'Accuracy': accuracies})

results_df = results_df.sort_values(by='Accuracy', ascending=False)

top_n = 5
plt.figure(figsize=(12, 6))
ax = sns.barplot(x='Accuracy', y='Hyperparameters', data=results_df.head(top_n), orient='h')

for index, value in enumerate(results_df.head(top_n)['Accuracy']):
    ax.text(value, index, f'{value:.4f}', ha='left', va='center', color='black')

plt.title(f'Top {top_n} Accuracy Across Different Hyperparameter Values')
plt.xlabel('Accuracy')
plt.ylabel('Hyperparameters')
plt.show()


# ### 4.3. Examining the performance of the best model

# Using the best hyperparameters obtained from a grid search, metrics such as accuracy, precision, recall, and F1 score are computed and presented, along with a heatmap visualization o

# In[ ]:


best_hyperparameters = results_df.iloc[0]['Hyperparameters']

print(f"Best Hyperparameters: "
      f"weights={best_hyperparameters[0]}, "
      f"p={best_hyperparameters[1]}, "
      f"n_neighbors={best_hyperparameters[2]}, "
      f"metric={best_hyperparameters[3]}, "
      f"leaf_size={best_hyperparameters[4]}, "
      f"algorithm={best_hyperparameters[5]}")


# In[66]:
```

```
best_knn = KNeighborsClassifier(
    weights=best_hyperparameters[0],
    p=best_hyperparameters[1],
    n_neighbors=best_hyperparameters[2],
    metric=best_hyperparameters[3],
    leaf_size=best_hyperparameters[4],
    algorithm=best_hyperparameters[5]
)

best_knn.fit(X_train, y_train)

y_pred = best_knn.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

print(f'Best Model Accuracy on Test Set: {accuracy:.4f}')
print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1 Score: {f1:.4f}')

conf_matrix = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(4, 3))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()


# ## 5. Model Training (Iterative Search)

# ### 5.1. Calling the helper functions for model training

# Defining functions for KNN cross-validation, training on folds, and finding optimal hyperparameter values through averaging test accuracies.

# In[41]:


def knn_cross_val_fold(X_train_fold, y_train_fold, X_val_fold, y_val_fold, knn):
    knn.fit(X_train_fold, y_train_fold)

    y_train_fold_pred = knn.predict(X_train_fold)
    y_val_fold_pred = knn.predict(X_val_fold)

    fold_train_accuracy = accuracy_score(y_train_fold, y_train_fold_pred)
    fold_test_accuracy = accuracy_score(y_val_fold, y_val_fold_pred)

    return fold_train_accuracy, fold_test_accuracy

def knn_cross_val(X_train, y_train, knn, kf=kf):
    fold_train_accuracies = []
    fold_test_accuracies = []

    for train_index, test_index in kf.split(X_train):
        X_train_fold, X_val_fold = X_train[train_index], X_train[test_index]
        y_train_fold, y_val_fold = y_train[train_index], y_train[test_index]

        fold_train_accuracy, fold_test_accuracy = knn_cross_val_fold(X_train_fold, y_train_fold, X_val_fold, y_val_fold, knn)

        fold_train_accuracies.append(fold_train_accuracy)
        fold_test_accuracies.append(fold_test_accuracy)

    avg_train_accuracy = np.mean(fold_train_accuracies)
    avg_test_accuracy = np.mean(fold_test_accuracies)

    return avg_train_accuracy, avg_test_accuracy

def find_best_value(test_accuracies, values_range):
    best_index = np.argmax(test_accuracies)
    best_value = values_range[best_index]
    return best_index, best_value


# ### 5.2. Nearest Neighbors Hyperparameter Tuning Curve

# Iterating KNN models with different neighbor values (3 to 29) using 10-fold cross-validation.

# In[42]:


knn = KNeighborsClassifier()

n_neighbors_values = [3, 5, 7, 9, 11, 13, 15]

train_accuracies = []
test_accuracies = []

for n_neighbors in n_neighbors_values:
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_index, best_n_neighbors = find_best_value(test_accuracies, n_neighbors_values)

plt.plot(n_neighbors_values, train_accuracies, label='Average Training Accuracy')
plt.plot(n_neighbors_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_n_neighbors, test_accuracies[best_index], color='red', marker='x', label=f'Best Test Accuracy (n_neighbors={best_n_neighbors})')
plt.xlabel('Number of Neighbors (n_neighbors)')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different n_neighbors Values with 3-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best n_neighbors: {best_n_neighbors}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_index]}")


# ### 5.3. Weights Methods Hyperparameter Tuning Curve

# Iterating KNN models with different distance weights methods (uniform and distance) using 10-fold cross-validation.

# In[43]:


weights_values = ['uniform', 'distance']
```

```
train_accuracies = []
test_accuracies = []

for weights in weights_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=weights)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_weights_index, best_weights = find_best_value(test_accuracies, weights_values)

plt.plot(weights_values, train_accuracies, label='Average Training Accuracy')
plt.plot(weights_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_weights, test_accuracies[best_weights_index], color='red', marker='x', label=f'Best Test Accuracy (weights={best_weights})')
plt.xlabel('Weights')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Weights Values with 3-Fold Cross-Validation')
plt.legend()
plt.show()

\print(f"Best weights: {best_weights}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_weights_index]}")


# ### 5.4. Minkowski distance p values Hyperparameter Tuning Curve

# Iterating KNN models with different p values for minkowski distance (0.5 to 4.0) using 10-fold cross-validation.

# In[44]:


p_values = [i/2 for i in range(2, 9)]

train_accuracies = []
test_accuracies = []

for p in p_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=p, metric='minkowski')
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_p_index, best_p = find_best_value(test_accuracies, p_values)

plt.plot(p_values, train_accuracies, label='Average Training Accuracy')
plt.plot(p_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_p, test_accuracies[best_p_index], color='red', marker='x', label=f'Best Test Accuracy (p={best_p})')
plt.xlabel('p')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different p Values with 3-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best p value: {best_p}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_p_index]}")


# ### 5.5. Spatial Algorithms Hyperparameter Tuning Curve

# Iterating KNN models with different spacial algorithm (auto, ball_tree, kd_tree, brute) using 10-fold cross-validation.

# In[45]:


algorithm_values = ['auto', 'brute']

train_accuracies = []
test_accuracies = []

for algorithm in algorithm_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=best_p, algorithm=algorithm)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_algorithm_index, best_algorithm = find_best_value(test_accuracies, algorithm_values)

plt.plot(algorithm_values, train_accuracies, label='Average Training Accuracy')
plt.plot(algorithm_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_algorithm, test_accuracies[best_algorithm_index], color='red', marker='x', label=f'Best Test Accuracy (algorithm={best_algorithm})')
plt.xlabel('Algorithm')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Algorithm Values with 3-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best algorithm: {best_algorithm}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_algorithm_index]}")


# ### 5.6. Leaf Size Values Hyperparameter Tuning Curve

# Iterating KNN models with different leaf size values (10 to 40) using 10-fold cross-validation.

# In[46]:


leaf_size_values = [10, 20, 30, 40]

train_accuracies = []
test_accuracies = []

for leaf_size in leaf_size_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=best_p, algorithm=best_algorithm, leaf_size=leaf_size)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_leaf_size_index, best_leaf_size = find_best_value(test_accuracies, leaf_size_values)

plt.plot(leaf_size_values, train_accuracies, label='Average Training Accuracy')
plt.plot(leaf_size_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_leaf_size, test_accuracies[best_leaf_size_index], color='red', marker='x', label=f'Best Test Accuracy (leaf_size={best_leaf_size})')
plt.xlabel('Leaf Size')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Leaf Size Values with 3-Fold Cross-Validation')
plt.legend()
plt.show()
```

```
print(f"Best leaf_size: {best_leaf_size}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_leaf_size_index]}")


# ### 5.7. Distance Metric Hyperparameter Tuning Curve

# Iterating KNN models with different distance metric (euclidean, manhattan, minkowski with best_p value) using 10-fold cross-validation.

# In[47]:


metric_values = ['euclidean', 'manhattan', 'minkowski']

train_accuracies = []
test_accuracies = []

kf = KFold(n_splits=10, shuffle=True, random_state=42)

for metric in metric_values:
    knn = KNeighborsClassifier(n_neighbors=best_n_neighbors, weights=best_weights, p=best_p, algorithm=best_algorithm, leaf_size=best_leaf_size, metric=metric)
    avg_train_accuracy, avg_test_accuracy = knn_cross_val(X_train, y_train, knn, kf=kf)
    train_accuracies.append(avg_train_accuracy)
    test_accuracies.append(avg_test_accuracy)

best_metric_index = np.argmax(test_accuracies)

best_metric = metric_values[best_metric_index]

best_metric_index, best_metric = find_best_value(test_accuracies, metric_values)

plt.plot(metric_values, train_accuracies, label='Average Training Accuracy')
plt.plot(metric_values, test_accuracies, label='Average Test Accuracy')
plt.scatter(best_metric, test_accuracies[best_metric_index], color='red', marker='x', label=f'Best Test Accuracy (metric={best_metric})')
plt.xlabel('Metric')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Metric Values with 3-Fold Cross-Validation')
plt.legend()
plt.show()

print(f"Best metric: {best_metric}")
print(f"Corresponding Test Accuracy: {test_accuracies[best_metric_index]}")


# ## 6. PCA Performance Trade-off

# Assessing PCA performance with varying number of components in a KNN model. It evaluates accuracy and records execution time, plotting results to evaluate trade-offs between accuracy

# In[61]:


n_components_values = list(range(1, 31))

accuracy_scores = []
execution_times = []

X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

for n_components in n_components_values:
    pca = PCA(n_components=n_components)
    knn_pca = KNeighborsClassifier(
    algorithm=best_hyperparameters[0],
    leaf_size=best_hyperparameters[1],
    metric=best_hyperparameters[2],
    n_neighbors=best_hyperparameters[3],
    p=best_hyperparameters[4],
    weights=best_hyperparameters[5])
    pipeline = Pipeline([
        ('scaler', StandardScaler()),
        ('pca', pca),
        ('knn', knn_pca)
    ])

    start_time = time.time()

    scores = cross_val_score(pipeline, X_train, y_train, cv=kf, scoring='accuracy')

    accuracy_scores.append(np.mean(scores))
    execution_times.append(time.time() - start_time)

fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.set_xlabel('n_components')
ax1.set_ylabel('Accuracy', color=color)
ax1.plot(n_components_values, accuracy_scores, color=color, marker='o')
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()
color = 'tab:blue'
ax2.set_ylabel('Execution Time (seconds)', color=color)
ax2.plot(n_components_values, execution_times, color=color, marker='x')
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()
plt.title('PCA Performance for Different n_components Values')
plt.show()
```

4) Neural Network: Steps to run the code:

1) Run the file `main.py` in the directory FashionMNIST/Neural_Network
2) Run the file `plot.py` in the directory FashionMNIST/Neural_Network

Script main.py in the directory Fashion_MNIST/Neural_Network

```
import ftns as ftns
import numpy as np
import torchvision
import torchvision.transforms as transforms
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import torch
from torch.utils.data import TensorDataset, DataLoader
import torch.nn as nn
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns



def main():
 X_train, y_train, X_test, y_test, X_val, y_val = ftns.load_fashion_dataset()

 X_train = np.load('X_train.npy')
 y_train = np.load('y_train.npy')
 X_test = np.load('X_test.npy')
 y_test = np.load('y_test.npy')
 X_val = np.load('X_val.npy')
 y_val = np.load('y_val.npy')

 print('X_train shape:', X_train.shape)
 print('y_train shape:', y_train.shape)
 print('X_val shape:', X_val.shape)
 print('y_val shape:', y_val.shape)
 print('X_test shape:', X_test.shape)
 print('y_test shape:', y_test.shape)
 print()


 train_data_tensor = torch.tensor(X_train, dtype=torch.float32)
 train_labs_tensor = torch.tensor(y_train, dtype=torch.long)
 train_dataset = TensorDataset(train_data_tensor, train_labs_tensor)

 val_data_tensor = torch.tensor(X_val, dtype=torch.float32)
 val_labs_tensor = torch.tensor(y_val, dtype=torch.long)
 val_dataset = TensorDataset(val_data_tensor, val_labs_tensor)

 test_data_tensor = torch.tensor(X_test, dtype=torch.float32)
 test_labs_tensor = torch.tensor(y_test, dtype=torch.long)
 test_dataset = TensorDataset(test_data_tensor, test_labs_tensor)

 batch_size = 64
 num_classes = 10
 learning_rate = 0.001
 num_epochs = 10

 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


 train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
 val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
 test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)



# Citation:
# Writing LeNet5 from Scratch in PyTorch, https://blog.paperspace.com/writing-lenet5-from-scratch-in-python/.



 class LeNet5(nn.Module):
  def __init__(self, num_classes):
   super(LeNet5, self).__init__()
   self.layer1 = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=0),
    nn.BatchNorm2d(6),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2))
   self.layer2 = nn.Sequential(
    nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
    nn.BatchNorm2d(16),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2))
   self.fc = nn.Linear(400, 120)
   self.relu = nn.ReLU()
   self.fully_conn1 = nn.Linear(120, 84)
   self.relu1 = nn.ReLU()
   self.fully_conn2 = nn.Linear(84, num_classes)

  def forward(self, x):
   out = self.layer1(x)
   out = self.layer2(out)
   out = out.view(out.size(0), -1)
   out = self.fc(out)
   out = self.relu(out)
   out = self.fully_conn1(out)
   out = self.relu1(out)
   out = self.fully_conn2(out)
   return out
```

```
model = LeNet5(num_classes).to(device)

criterion = nn.CrossEntropyLoss()

optim1 = torch.optim.Adam(model.parameters(), lr=learning_rate)


train_loss_lst = []
val_loss_lst = []
train_acc_lst = []
val_acc_lst = []


for epoch in range(num_epochs):
 train_all_loss = 0
 val_all_loss = 0

 train_fine = 0
 val_fine = 0

 train_all = 0
 val_all = 0

 for i, (inputs, labs) in enumerate(train_loader):
  inputs = inputs.to(device).unsqueeze(1)
  labs = labs.to(device)
  outputs = model(inputs)
  loss = criterion(outputs, labs)
  optim1.zero_grad()
  loss.backward()
  optim1.step()
  train_all_loss += loss.item()
  _, pred = torch.max(outputs.data, 1)
  train_all += labs.size(0)
  train_fine += (pred == labs).sum().item()



 with torch.no_grad():
  for i, (inputs, labs) in enumerate(val_loader):
   inputs = inputs.to(device).unsqueeze(1)
   labs = labs.to(device)
   outputs = model(inputs)
   loss = criterion(outputs, labs)
   val_all_loss += loss.item()
   _, pred = torch.max(outputs.data, 1)
   val_all += labs.size(0)
   val_fine += (pred == labs).sum().item()



 train_normalized_loss = train_all_loss / len(train_loader.dataset)
 train_loss_lst.append(train_normalized_loss)
 np.save('train_loss_arr.npy', np.array(train_loss_lst))
 val_normalized_loss = val_all_loss / len(val_loader.dataset)
 val_loss_lst.append(val_normalized_loss)
 np.save('val_loss_arr.npy', np.array(val_loss_lst))

 print('epoch', epoch)
 print('train_normalized_loss', train_normalized_loss)
 print('val_normalized_loss', val_normalized_loss)
 print('train accuracy', train_fine/train_all)
 print('Validation accuracy', val_fine/val_all)
 print()

 train_acc_lst.append(train_fine/train_all)
 np.save('train_acc_arr.npy', np.array(train_acc_lst))
 val_acc_lst.append(val_fine/val_all)
 np.save('val_acc_arr.npy', np.array(val_acc_lst))


all_labs = []
all_predictions = []


cv_mean_acc, cv_error_bar = ftns.cross_validate(train_loader, num_classes, num_epochs, batch_size, device, learning_rate, num_epochs, criterion)



with torch.no_grad():
 fine = 0
 all1 = 0
 for inputs, labs in test_loader:
  inputs = inputs.to(device).unsqueeze(1)
  labs = labs.to(device)
  outputs = model(inputs)
  _, pred = torch.max(outputs.data, 1)
  all1 += labs.size(0)
  fine += (pred == labs).sum().item()
  #Chatgpt prompt: How to plot a confusion matrix given labels and groundtruths from a for loop.
  all_labs.extend(labs.cpu().numpy())
  all_predictions.extend(pred.cpu().numpy())
```

```
  testing_accuracy =  fine / all1
  print('testing_accuracy', testing_accuracy)
  print()



 cm = confusion_matrix(all_labs, all_predictions)
 plt.figure(figsize=(10, 7))
 ax = sns.heatmap(cm, annot=True, fmt='g',  cbar=False)
 plt.title('Confusion Matrix')
 plt.xlabel('pred labs')
 plt.ylabel('True labs')
 plt.savefig('confusion_matrix.png')




 np.save('final testing accuracy', testing_accuracy)
 np.save('cv_mean_acc', cv_mean_acc)
 np.save('cv_error_bar', cv_error_bar )



 return 1

main()
```

Script plot.py in the directory Fashion_MNIST/Neural_Network

```python
import numpy as np
import matplotlib.pyplot as plt


def plot_array(arr1, label1, arr2, label2, title, xlabel, ylabel, fig_name):
    index = np.arange(1, len(arr1) + 1)
    plt.figure(figsize=(10, 6))
    plt.plot(index, arr1, '-o', color = 'orange' ,label=label1)
    plt.plot(index, arr2, '-o' ,label=label2)
    plt.legend()
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.grid(True)
    plt.savefig(fig_name)
    plt.show()




train_loss = np.load('train_loss_arr.npy')
val_loss = np.load('val_loss_arr.npy')

train_acc = np.load('train_acc_arr.npy')
val_acc = np.load('val_acc_arr.npy')

plot_array(train_loss, 'Training Loss', val_loss, 'Validation Loss', 'Training and Validation loss vs Epochs', 'Epochs', 'Cross Entropy Loss', 'loss_vs_epochs_fashionmnist.png')
plot_array(train_acc, 'Training Accuracy', val_acc, 'Validation Accuracy', 'Training and Validation Accuracy vs Epochs', 'Epochs', 'Accuracy', 'accuracy_vs_epochs_fashionmnist.png')
```

Script ftns.py in the directory Fashion_MNIST/Neural_Network

```python
import ftns as ftns
import numpy as np
import torchvision
import torchvision.transforms as transforms
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
from torch.utils.data import TensorDataset, DataLoader


def load_fashion_dataset():

    transform = transforms.Compose([
        transforms.Resize((32, 32)),
        transforms.ToTensor()
    ])

    train_dataset = torchvision.datasets.FashionMNIST(
        root='./data',
        train=True,
        transform=transform,
        download=True
    )

    test_dataset = torchvision.datasets.FashionMNIST(
        root='./data',
        train=False,
        transform=transform,
        download=True
    )



    X_train_transformed = []
    for i in range(len(train_dataset)):
        image, _ = train_dataset[i]
        image_np = image.numpy()
        X_train_transformed.append(image_np)
    X_train_transformed = np.array(X_train_transformed)

    X_test_transformed = []
    for i in range(len(test_dataset)):
        image, _ = test_dataset[i]
        image_np = image.numpy()
        X_test_transformed.append(image_np)
    X_test_transformed = np.array(X_test_transformed)


    X_train = X_train_transformed
    y_train = train_dataset.targets.numpy()

    X_train = X_train.reshape((X_train.shape[0], -1))
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_train, X_val, y_train, y_val = train_test_split(
        X_train_scaled, y_train, test_size=0.3, random_state=30)

    X_test = X_test_transformed
    y_test = test_dataset.targets.numpy()
    X_test = X_test.reshape((X_test.shape[0], -1))
    X_test_scaled = scaler.transform(X_test)

    np.save('X_train.npy', X_train.reshape(X_train.shape[0], 32, 32))
    np.save('y_train.npy' , y_train)
    np.save('X_test.npy' ,X_test_scaled.reshape(X_test_scaled.shape[0], 32, 32))
    np.save('y_test.npy' , y_test)
    np.save('X_val.npy', X_val.reshape(X_val.shape[0], 32, 32))
    np.save('y_val.npy', y_val)

    return X_train.reshape(X_train.shape[0], 32, 32), y_train, X_test_scaled.reshape(X_test_scaled.shape[0], 32, 32), y_test, X_val.reshape(X_val.shape[0], 32, 32), y_val

# Citation:
# Writing LeNet5 from Scratch in PyTorch, https://blog.paperspace.com/writing-lenet5-from-scratch-in-python/.



class LeNet5(nn.Module):
    def __init__(self, num_classes):
        super(LeNet5, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=0),
            nn.BatchNorm2d(6),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.fc = nn.Linear(400, 120)
        self.relu = nn.ReLU()
        self.fully_conn1 = nn.Linear(120, 84)
        self.relu1 = nn.ReLU()
        self.fully_conn2 = nn.Linear(84, num_classes)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        out = self.relu(out)
        out = self.fully_conn1(out)
        out = self.relu1(out)
        out = self.fully_conn2(out)
        return out
```

```python
def cross_validate(val_loader, num_classes, total_epochs, batch_size, device, learning_rate, num_epochs, cost):
    # Chatgpt prompt: How to implement k-fold cross-validation for my pyTorch deep learning model
    kf = KFold(n_splits=5, shuffle=True, random_state=1)

    accuracies = []

    for fold, (train_idx, val_idx) in enumerate(kf.split(np.arange(len(val_loader.dataset)))):
        train_subsampler = torch.utils.data.SubsetRandomSampler(train_idx)
        val_subsampler = torch.utils.data.SubsetRandomSampler(val_idx)

        train_fold_loader = DataLoader(val_loader.dataset, batch_size=batch_size, sampler=train_subsampler)
        val_fold_loader = DataLoader(val_loader.dataset, batch_size=batch_size, sampler=val_subsampler)


        model = LeNet5(num_classes).to(device)
        optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


        for epoch in range(num_epochs):
            train_all_loss = 0
            train_fine = 0
            train_all = 0
            for i, (inputs, labs) in enumerate(train_fold_loader):
                inputs = inputs.to(device).unsqueeze(1)
                labs = labs.to(device)
                outputs = model(inputs)
                loss = cost(outputs, labs)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
                train_all_loss += loss.item()
                _, pred = torch.max(outputs.data, 1)
                train_all += labs.size(0)
                train_fine += (pred == labs).sum().item()


        model.eval()
        fine, total = 0, 0
        with torch.no_grad():
            for inputs, labs in val_fold_loader:
                inputs = inputs.to(device).unsqueeze(1)
                labs = labs.to(device)
                outputs = model(inputs)
                _, pred = torch.max(outputs.data, 1)
                total += labs.size(0)
                fine += (pred == labs).sum().item()

        fold_accuracy = fine / total
        accuracies.append(fold_accuracy)
        print(f'Fold {fold+1}, Accuracy: {fold_accuracy:.4f}')

        std_dev = np.std(accuracies, ddof=1)
        std_err = std_dev / np.sqrt(len(accuracies))



    print(f'Average Cross-Validation Accuracy: {np.mean(accuracies):.4f}')
    return np.mean(accuracies), std_err
```