Comparison of Machine Learning Methods on Tabular and Numerical Datasets

1st Max Gao
School of Manufacturing Networks and Systems
Arizona State University
Tempe, Arizona, United States of America
mgao18@asu.edu

2nd Yuanhao Chen
School for Engineering of Matter, Transport and Energy
Arizona State University
Tempe, Arizona, United States of America
yche1066@asu.edu

3rd Sai Manish Rao Marru
School of Computing and Augmented Intelligence
Arizona State University
Tempe, Arizona, United States of America
smarru@asu.edu

4th Anaam Mostafiz
School for Engineering of Matter, Transport and Energy
Arizona State University
Tempe, Arizona, United States of America
amostafi@asu.edu

I. Introduction

Machine Learning is the study of data-driven prediction and analysis, built upon fundamental principles of statistics and probability theory. Machine Learning represents the translation of complex tasks, such as the classification of spam emails from legitimate emails, into a form that is mathematically executable, generally using computers. As the development and evolution of machine learning models is fundamentally based exclusively on data, such translations of complex tasks becomes essentially an approximation attempting to replicate results, with accuracy being limited by the size of the data available. [1]

With Machine Learning appearing in many different forms, it becomes important to gain an understanding of which of these forms, and more so the hyperparameters that govern them, are most important for analyzing different datasets. It is the goal of this study to investigate the usability of four such classifiers on two different data sets, to determine the particular properties of each classifier and the particular impact of their hyperparameters.

A. Datasets Used

In this study, three datasets will be analyzed by various Machine Learning classifiers. The first, the Wisconsin Breast Cancer Diagnostic Dataset, consists of thirty numerical features and a binary diagnostic of whether a present tumor is benign or malignant. [2] The second, UCI Adult, is a similar dataset consisting of categorical features with an observation indicating annual income being above or below $50,000. [3] Lastly, Fashion MNIST is a numerical dataset of many low-resolution images presenting one of ten apparel items. [4]

B. Model Classes Used

Four classification models will be used in this study, each having similar applications but differing approaches. Firstly, Logistic Regression is a binary classification model that estimates the probability of a particular binary outcome using a logit transformation. The outcome is the log-odds, or the log of the probability of one state over another. [5] Next, Support Vector Machines (SVM), is a class-separator technique that seeks to maximize the margin, or separation between two classes in a data space. [5] Third, k-Nearest Neighbor (k-NN), is a simple classifier that assigns a class to a new data point based on the classes of the nearest k data points. [5] Due to known instability issues with working in high-dimensional data spaces, this will be paired with Principle Component Analysis (PCA), which is a data simplification technique that compresses the dimensionality of complex datasets. [5] Finally, the Neural Network is a versatile classifier featuring many possible layers of data transformation. If working on images, a Convolution filter set can also be applied to pre-transform the image-based data to a format more suitable for a standard Neural Network.

C. Team Contribution

The four team members of this project study contributed in different ways towards its success shows in Table I below.

Team Member	Max Gao	Anaam Mostafiz	Yuanhao Chen	Sai Manish Rao Marru
Contribution	Neural Networks and Report Writing	Logistic Regression and Report Writing	k-NN / PCA and Report Writing	SVM and Report Writing

TABLE I: Team Member Contributions

II. Dataset 1: Diagnostic Breast Cancer

A. General Approach

Before implementing all models, the Breast Cancer dataset was preprocessed using functions from the scikit-learn library. [6] This involved using the StandardScaler() function to standardize the numerical features, and train_test_split() to partition the dataset into 80% training and 20% testing. The numpy library was used to aid in basic array operations. [7]

All hyperparameter tuning was done using grid search, using either scikit-learn's GridSearchCV or manually with itertools. [8] Where applicable, a 5-fold cross-validation structure was implemented using KFold() from scikit-learn. Lastly, the libraries pandas, matplotlib, and seaborn were used for data recording and visualization. [9]–[11] Most of the code was written by the team, with some general coding assistance from ChatGPT. [12]

B. Logistic Regression Setup and Results

Overall, the logistic regression modeling procedure is as follows. First, the data is split into training and test sets. The Fashion MNIST data already imports into training and test sets, while the other datasets are split into 80% training and 20% test data. Then, 5-fold cross-validation is performed on the training set for a parameter grid. Finally, the parameters with the lowest average validation error are used to train the model and evaluate its accuracy with the test set.

Logistic regression is performed with the LogisticRegression() function, and k-fold cross-validation of the parameter grid is performed using the GridSearchCV() function. This procedure is repeated for logistic regression with L2, L1, both L2 and L1 (Elastic-Net), and no regularization to compare their accuracy. All parameters can be found in Table II.

Reg. Type	**Breast Cancer and Adult Datasets**		**Fashion MNIST Dataset**	
	Parameters	**Values**	**Parameters**	**Values**
L2	C: Regularization Parameter	[0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 100]	C: Regularization Parameter	[0.01, 0.1, 1]
	C: Regularization Parameter	[0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 100]	C: Regularization Parameter	[0.01, 0.1, 1]
L1	C: Regularization Parameter	[0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 100]	C: Regularization Parameter	[0.01, 0.1, 1]
	C: Regularization Parameter	[0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 100]	C: Regularization Parameter	[0.01, 0.1, 1]
Elastic-Net	C: Regularization Parameter	[0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 100]	C: Regularization Parameter	[0.01, 0.1, 1]
	`l1_ratio`: Mixing parameter	[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]	`l1_ratio`: Mixing parameter	[0.2, 0.4, 0.6, 0.8]
None	n/a	n/a	n/a	n/a

TABLE II: Logistic Regression Parameters for 3 Datasets

For the Breast Cancer dataset, L1 regularization with C=1 and Elastic-Net regularization with C=1 and `l1_ratio`=0.8 both resulted in the best average validation accuracy of 0.978. Since Elastic-Net was more accurate than L1 regularization for most other values of C, it seems that Elastic-Net is the better overall model. Since `l1_ratio`=0.8, this suggests that the relevant features are sparse and that there are some correlated features.

With this model, the test set confusion matrix is [70, 1; 2, 41]. This suggests that the model is a great predictor of both class 0 and class 1, which is essential when predicting cancer. The test set classification report corroborates this conclusion, which can be found in Table X and Fig. 5.

C. Support Vector Machines Setup and Results

Prior to SVM model training, data preprocessing is conducted to address missing values and normalize feature ranges. The baseline SVM model is established using *SVC()* with default settings *(default parameters: kernel:rbf, C:1.0, γ:scale)*.

Hyperparameter tuning is performed via *GridSearchCV()*, with parameters specified in Table IX, employing a 5-fold cross-validation *(training-validation split ratio* $80\% - 20\%$*)* to find the parameter set that maximizes mean accuracy. A subsequent round of tuning is conducted reduced search space for parameters *(C,γ,d)*, based on the kernel type. Performance comparisons between the tuned model and the baseline are drawn using classification reports, validation curves, and a confusion matrix.

The range of parameter values that have been evaluated are listed in the Table XII *(total candidates evaluated: 5202)* and Table XIII *(total candidates evaluated: 10,000)* and the best parameters obtained are *(kernel: RBF, C: 5.0802, γ: 0.0171)*. By tuning the parameters SVM model was able to achieve an *accuracy of* 98% in comparison to the default/baseline model of *accuracy of* 97% (Table XI). With the help of validation curves for C and γ (Fig 6 and Fig 7), we can observe that at *(C: 5.0802, γ: 0,0171)* both training and test accuracy are high indicating that the model is neither underfitting nor overfitting.

In case of predicting medical cancer, it is important to reduce false positive and false negative predictions, the SVM model was able to do this and with the help of confusion matrix, we can see that *(Number of False positives: 0, Number of False Negatives: 2)*. And also, the F1-Score of *(Class Benign: 98% and Class Malignant: 98%)*, indicate that both the classes are balanced and the model performs well for both the class.

SVM Kernels and its Parameters details: Appendix Section VII-A, Table IX
Details related to Hyperparameter tuning: Appendix Section VII-C, (Table XII and Table XIII)
Validation Curves for Parameters C and γ: Appendix Section VII-C, (Fig 6 and Fig 7)
SVM model comparision and Classification Report: Section VII-C, Table XI

Confusion matrix:

$$\begin{pmatrix} 70 & 0 \\ 2 & 42 \end{pmatrix}$$

D. k-Nearest Neighbors Setup and Results

The PCA + k-NN method can efficiently simplify complex data while maintaining critical features for accurate predictive modeling. The PCA was applied to reduce the dimensionality of the datasets, and we chose a variance threshold of 95% for it, ensuring that the majority of the data's variability was retained while simplifying the models. After dimensionality reduction, we employed the k-NN algorithm for classification. We conducted hyperparameter tuning, particularly adjusting the $n\_neighbors$ parameter, to optimize the classifier's performance for each dataset. All hyperparameters are included in Table III below. Finally, we evaluated the model's performance using cross-validation, and identified the optimal parameters through a grid search.

Hyperparameter	**Values / Range**
k-NN Neighbors	{3, 5, 7, 9, 11}
PCA Variance	95%

TABLE III: Hyperparameters Evaluation in PCA + k-NN Model

Particularly, variable y is expected to be a one-dimensional array of shape $n\_samples$, which holds the target values (labels) for the classification or regression tasks. The $y\_reshaped$ line checks if y is a pandas DataFrame. If yes, then it will convert y to a one-dimensional numpy array, which is necessary to be processed here.

The procedure began with a standardized scaling of the dataset, followed by PCA for dimensionality reduction. The k-NN classifier was then set up with a hyperparameter grid that defined the range of neighbors to consider. Through a cross-validation approach with a 5-fold split, we sought the optimal number of neighbors, which was determined to be 3, as per the grid search results. For k-NN neighbors we tuned from the set {3, 5, 7, 9, 11}.

Our validation approach involves a training-validation split (70%-30%), ensuring a clear and consistent evaluation process. The performance of the model was primarily assessed using precision (F1-score metrics, indicates the model's ability to classify the different classes accurately). The best parameters obtained from hyperparameter tuning were $n\_neighbors$: 3, and the classification performance on the test set has an accuracy of 95%, as well as others all being 95%. This indicates a well-balanced model that performs equally well for both classes (B for benign and M for malignant). The training time for the model was a brief 0.56 seconds, emphasizing an efficient computational process. The results, including the detailed classification report and the confusion matrix, are attached in the Table XIV, which demonstrates the model's predictive capabilities.

E. Neural Networks Setup and Results

For training and evaluating neural networks, the library TensorFlow [13] was used in conjunction with utility libraries such as Multiprocessing, CSV, and itertools for easy mass data collection. The architecture of the script was built to accommodate for the requirements for multi-threading, with a single model creation and evaluation function being executed by a pool of asynchronous workers using Multiprocessing's Pool() and starmap() functions as can be seen in Appendix Sections VII-O, VII-S, and VII-W. A grid search was done over four hyperparameters, with every hyperparameter combination being trained over 100 epochs and evaluated with 5-fold cross validation. Validation loss, accuracy, and training time were recorded.

The four hyperparameters varied in the search were hidden layer width, hidden layer count, batch size, and optimizer function. The variances of these hyperparameters were anticipated to be the most impactful to score. With a small dataset implying fast computation time, at least five potential values of layer width, batch size, and optimizer function were tested. Only two values for hidden layer count were considered, as three or more hidden layers would likely make for an unnecessarily complicated model. For simplicity, all activation functions save for the final one were fixed to be ReLU, and all hidden layers were made the same size.

After searching all combinations, it was found that most trained models were highly accurate, with 190 of 350 models returning a validation accuracy of better than 98%. The best selected model reached a validation accuracy of 98.9% and a test accuracy of 96.5%, implying that the validation results are representative of hypothetical test results on other combinations.

Among top models, actual hyperparameter values varied greatly, but it was found as in Figure 1 that a smaller batch size, greater layer depth, fewer layers, and usage of either SGD, adam, or rmsprop optimizers returned better results. This primarily indicates that a Neural Network model works best with one large and complicated hidden layer when working with this sort of data. It also seems that this is universal when training models with different batch sizes and using different optimizer functions. It should also be noted that the selected best model does not necessarily adhere to this trend, either, as it was trained with two hidden layers of depth 20, with a batch size of 2 on the adam optimizer. A more detailed analysis is available in Appendix VII-D's Table XV.

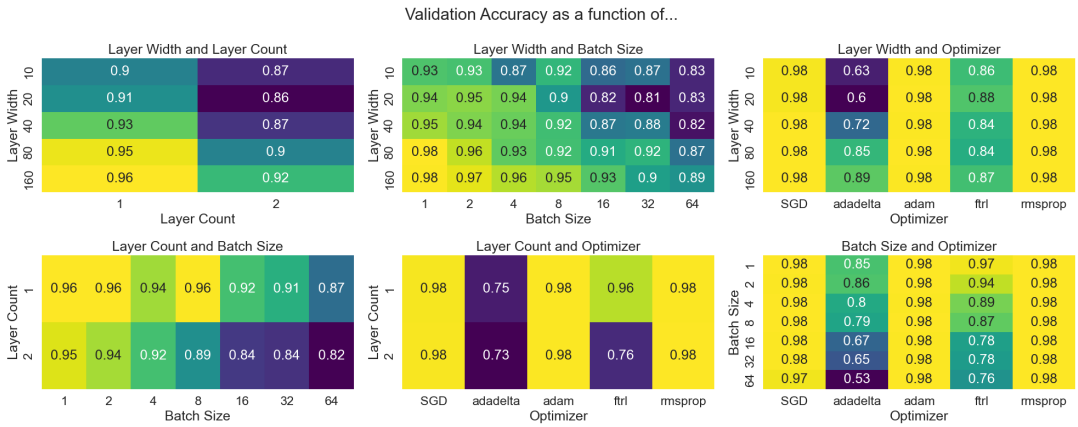

Fig. 1: Dataset 1 Hyperparameter Analysis based on Validation Accuracy

III. Dataset 2: Adult

A. General Approach

The overall plan for handling the UCI Adult dataset is very similar to that of the Breast Cancer dataset, with only the added step of using scikit-learn's OneHotEncoder(), ColumnTransformer(), and Pipeline() functions. In case of SVM, LableEncoder() is used to encode the categorical data. These are only used to convert the mixed tabular and categorical data format into a sparse matrix format that can be interpreted by other machine learning functions. If the dataset is downloaded, then the training and test sets are already provided, so spliting the entire data is not needed.

B. Results pertaining to Logistic Regression

For the Adult dataset, the best logistic regression model was found to be L1 regularization with C=0.01, with an average validation accuracy of 0.8203. Since it is L1 regularization, this suggests that the relevant features are sparse.

With this model, the test set confusion matrix is [6395, 350; 1272, 1028]. This suggests that the model is a good predictor of class 0, but not the best predictor of class 1. The test set classification report corroborates this conclusion.

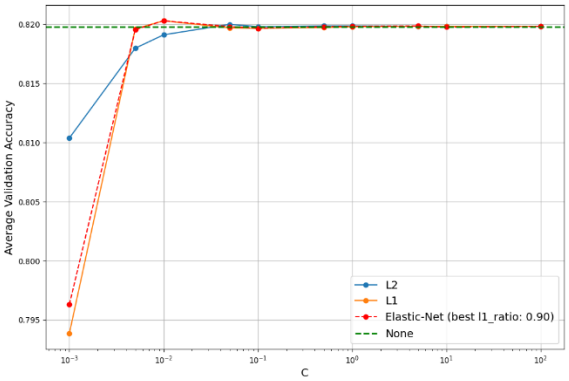

Fig. 2: Average Validation Accuracy vs C

Best parameters: {L1, C: 0.01}
Confusion matrix:

$$\begin{pmatrix} 6395 & 350 \\ 1272 & 1028 \end{pmatrix}$$

Classification report:

	Precision	Recall	F1-Score	Support
$Income \leq 50K$	0.83	0.95	0.89	6745
$Income > 50K$	0.75	0.45	0.56	2300
Accuracy	–	–	0.82	9045
Macro Avg.	0.79	0.70	0.72	9045
Weighted Avg.	0.81	0.82	0.80	9045

TABLE IV: Learning Result for Dataset 2

C. Results pertaining to Support Vector Machines

Similar to the SVM model procedure described in the Section II-C for the UCI Breast Cancer dataset, performed a hyperparameter tuning to find the optimal parameters on the given dataset with a 5-fold cross-validation *(training-validation split ratio* $80\% - 20\%$*)*. The range of parameter values that have been evaluated are listed in the Table XVI *(total candidates evaluated: 1072)* and Table XVII *(total candidates evaluated: 1000)* and the best parameters obtained for this search for parameters are *(kernel: RBF, C: 1.9067,* γ*: 0.1)*, as shown in the validation curves for C (Fig 9) and γ (Fig 8). And also the validation curves for C and γ indicates that at *(C: 1.9067,* γ*: 0.1)* the model neither underfits nor overfits, as both training and test accuracy are high.

Despite the hyperparameter tuning, the SVM model's accuracy is approximately 85%, which is nearly the same as the accuracy of the baseline/default model (Table V). This outcome can be observed from the confusion matrix, which indicates

that the model correctly predicts low-income instances $\leq 50K$ with high accuracy, but is less accurate with high-income $> 50K$ predictions, hinting at a potential class imbalance or other complexities in the data that may not be addressed by tuning alone. And also, the lower recall (56%) and F1-Score (64%) for ($Income > 50K$) class, as compared to the ($Income \leq 50K$) class (Table V), indicates that the classes are imbalanced and the model does not perform well on the class with ($Income > 50K$).

SVM Kernels and its Parameters details: Appendix Section VII-A, Table IX
Details related to Hyperparameter tuning: Appendix Section VII-F, (Table XVI and Table XVI)
Validation Curves for Parameters C and γ: Appendix Section VII-F, (Fig 8 and Fig 9)

Confusion matrix:

$$\begin{pmatrix} 10658 & 702 \\ 1621 & 2079 \end{pmatrix}$$

	Model with Tuned Parameters					**Baseline/Default Model**				
Parameters	$kernel$: RBF	C: 1.90679072		γ: 0.1		$kernel$: RBF	C: 1.0		γ: scale, $\frac{1}{n\_features \times \text{var}(X)}$	
Scoring Metrics		Precision	Recall	F1-Score	Support		Precision	Recall	F1-Score	Support
	$Income \leq 50K$	0.87	0.94	0.90	11360	$Income \leq 50K$	0.87	0.94	0.90	11360
	$Income > 50K$	0.74	0.56	0.64	3700	$Income > 50K$	0.75	0.55	0.64	3700
	Accuracy	–	–	0.85	15060	Accuracy	–	–	0.85	15060
	Macro Avg.	0.81	0.75	0.77	15060	Macro Avg.	0.81	0.75	0.77	15060
	Weighted Avg.	0.84	0.85	0.84	15060	Weighted Avg.	0.84	0.84	0.84	15060

TABLE V: SVM Model Evaluation based on Scoring Metrics for Dataset 2

D. Results pertaining to k-Nearest Neighbors

The confusion matrix now clearly distinguishes between the binary classes, showing a total of 9874 true positives for class '0' and 1969 true positives for class '1'. This indicates that the model is highly effective at identifying class '0' instances, with a high true positive rate of 92%. Conversely, the model exhibits a lower sensitivity towards class '1', with a notable 1585 instances that were false negatives, indicating that these instances were incorrectly classified as class '0'.

The Fig. 10 illustrates the proportion of variance explained by each PCA component. The sharp decline observed in the variance ratio remains consistent with the nature of PCA, where the initial components capture the most significant amount of information. The first component accounts for a substantial portion of the variance, emphasizing the importance of the leading components in the dimensionality reduction process.

The revised accuracy of the model stands at 83%, which suggests a competent level of classification performance. The macro-average F1-score, which considers the balance between precision and recall, is at 75%. This indicates a relatively balanced performance across the binary classes, taking into account the class imbalance. The training duration for this refined model increased slightly to 96.29 seconds, which is understandable given the additional preprocessing step and the complexity of the model. All these analysis are included in Table XVIII.

E. Results pertaining to Neural Networks

With insights from the Neural Network experiment on the first dataset and forecasted challenges involved in processing the larger UCI Adult dataset, changes to the planned hyperparameter search space were necessary. Firstly, with a much greater number of samples, raising the batch size was deemed appropriate to cut down on computation time, despite potential gains in accuracy otherwise. Larger layers were also explored due to the greater number of features in the sparse matrix format. Modifying the optimizer functions was considered, though ultimately decided against due to the change in feature format. Optimizers that performed well previously were not expected to do so in this experiment,

After a complete grid search across the same four parameters, the models trained using the UCI Adult Dataset were found to fare worse than the UCI Breast Cancer counterparts in terms of validation accuracy, despite access to much more data. While all 280 models were found to have validation accuracies of greater than 75%, the best model had a validation accuracy of only 87.7% and a corresponding 81.5% test accuracy. As seen in Appendix VII-G and Table XIX, there is a significant imbalance in performance in predicting labels, possibly caused by imbalance in the training data itself.

The same sensitivity analysis as shown in Figure 3 showed that large layer depths, two hidden layers, small batch sizes, and the adam optimizer returned the best results, but only by a slight amount. These relationships suggest very weak overall impact on validation accuracy due to hyperparameter tuning, requiring instead very precise combinations of parameters to see scores marginally better than baseline.

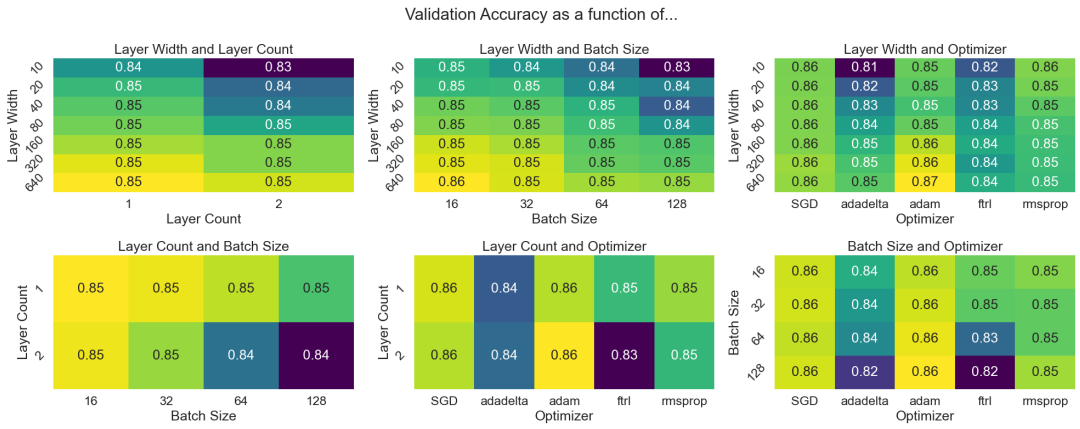

Fig. 3: Dataset 2 Hyperparameter Analysis based on Validation Accuracy

IV. Dataset 3: Fashion MNIST

A. General Approach

The Fashion MNIST datasets are acquired through the torchvision library [14], which are already separated into training and testing components. All data is then normalized by dividing the grayscale pixel values by 255, and then passed on to respective algorithms.

B. Results pertaining to Logistic Regression

Due to the high number of classes and samples in the Fashion MNIST data, it uses a less exhaustive parameter grid to reduce code runtime. All parameters can be found in Table II.

For the Fashion MNIST dataset, the best logistic regression model was found to be Elastic-Net regularization with C=0.1 and `l1_ratio`=0.2 (Fig: 11), with an average validation accuracy of 0.8575. Since `l1_ratio`=0.2, this suggests that there are many correlated features, and a few irrelevant features.

The test set confusion matrix (Fig: 12) suggests that the model is a good predictor in general of the classes, with the worst precision for class 6. It is a great predictor for classes 1, 5, 7, 8, and 9. The test set classification report corroborates this conclusion.

Category	0	1	2	3	4	5	6	7	8	9	Average
Precision	0.80	0.98	0.73	0.83	0.74	0.94	0.63	0.91	0.93	0.95	0.84
Recall	0.81	0.96	0.74	0.87	0.76	0.92	0.56	0.94	0.94	0.94	0.84
F1-Score	0.81	0.97	0.73	0.85	0.75	0.93	0.60	0.92	0.94	0.94	0.84
Support	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	10000

TABLE VI: Learning Result for Dataset 3

C. Results pertaining to Support Vector Machines

Due to the computational intensity of training SVM model on the FashionMNIST dataset *(each fit/training time*$\approx 10mins$*)*, reshaped the (28×28) image size to (14×14) without compromising the accuracy of the model *(each fit/training time*$\approx 3mins$*)* and performed a hyperparameter tuning to find the optimal parameters on the given dataset with a 5-fold cross-validation *(training-validation split ratio* $80\% - 20\%$*)*, similar to the SVM model procedure described in the Section II-C for the UCI Breast Cancer dataset. The range of parameter values that have been evaluated are listed in the Table XX *(total candidates evaluated: 301)* and Table XXI *(total candidates evaluated: 100)* and the best parameters obtained for this search for parameters are *(kernel: RBF, C: 4.354,* γ*: 0.1668)*, as shown in the validation curves for C (Fig 14) and γ (Fig 13). And also the validation curves for C and γ indicates that at *(C: 4.354,* γ*: 0.1668)* the model neither underfits nor overfits, as both training and test accuracy are high.

By tuning the parameters SVM model was able to achieve an *accuracy of* 90% in comparison to the default/baseline model of *accuracy of* 88% (Table VII). Based on the *F1 Score(*72%*), Table XXII* of class(6: Shirt) compared to the average *F1 Score(*90%*)* of all classes, indicates that class 6 can be imbalanced and the model may not perform well on classifying class-6.

SVM Kernels and its Parameters details: Appendix Section VII-A, Table IX
Details related to Hyperparameter tuning: Appendix Section VII-J, (Table XX and Table XX)
Validation Curves for Parameters C and γ: Appendix Section VII-J, (Fig 13 and Fig 14)
Classification Report: Appendix Section VII-A, Table XXII

	Model with Tuned Parameters	Baseline/Default Model
Parameters	$kernel$: RBF C: 4.35400465 γ: 0.16681005	$kernel$: RBF C: 1.0 γ: scale, $\frac{1}{n\_features \times \text{var}(X)}$
Accuracy	90%	88%
$BestAccuracy$: 90%		

TABLE VII: SVM Model Evaluation based on Scoring Metrics for Dataset 3

D. Results pertaining to k-Nearest Neighbors

The chosen hyperparameter for the k-NN algorithm is $k = 5$, which refers to the number of nearest neighbors to consider when making a prediction. The model's overall accuracy reached 86%, and the precision and recall metrics are balanced for most classes. For instance, class '1' (Trouser) has a high precision and recall of 99% and 97% respectively, indicate that the model is particularly adept at identifying this category with few false positives or negatives.

From the confusion matrix (Fig. 15 and Table XXIII), class '6' (Shirt) has a lower f1-score of 61%, suggesting that this class is the most challenging for the model to predict accurately, potentially due to similarity in features with other categories like Tops (class '0') and Coats (class '4'). Next, Fig. 16 cumulative variance plot reveals that to achieve the threshold of 95% explained variance, the model requires a substantial number of principal components. At last, the training time reported is 96.29 seconds.

E. Results pertaining to Neural Networks

In addition to the Neural Network structure established for the prior two datasets, a convolution and a maxpool layer were added to process and downscale the image before passing the refined data to the Feed-Forward Neural Network. Along with the four hyperparameters explored in past experiments, the number of filters in the convolution layer was also to be varied throughout grid search.

However, early testing found an extreme increase in processing time considering a large number of samples, high dimensionality even after maxpool, and the added step of convolution. Because of this, severe restrictions on search space had to be applied, seeing kernel size being locked to 3x3, and significant reductions in how many different hyperparameter values are tested. The number of epochs was also reduced from 100 to 30, accepting losses in accuracy. Despite this, The full list of five optimizer functions were kept as their impact on accuracy was then uncertain.

A comprehensive review of hyperparameter impact on validation accuracy is as follows in Figure 4. We notice that the preferences towards complex single-layer models also appears here, solidifying the trend across all three Neural Network classifiers. Seeing a lack of complex gradients, These plots also suggest little to no cross-correlation between hyperparameters in determining accuracy.

Lastly, as was done with both prior experiements, the best model hyperparameters were selected based on validation accuracy, which was found to be quite high at 96.3%. The test accuracy was comparatively reasonable at 91.1%. With ten labels able to be explored, a confusion matrix was created in Appendix VII-K as Figure 17 to determine which were especially troublesome. It appears that while the model is very accurate overall, a significant number of samples belonging to label 6 are being mistaken for samples belonging to label 0, and vise versa.

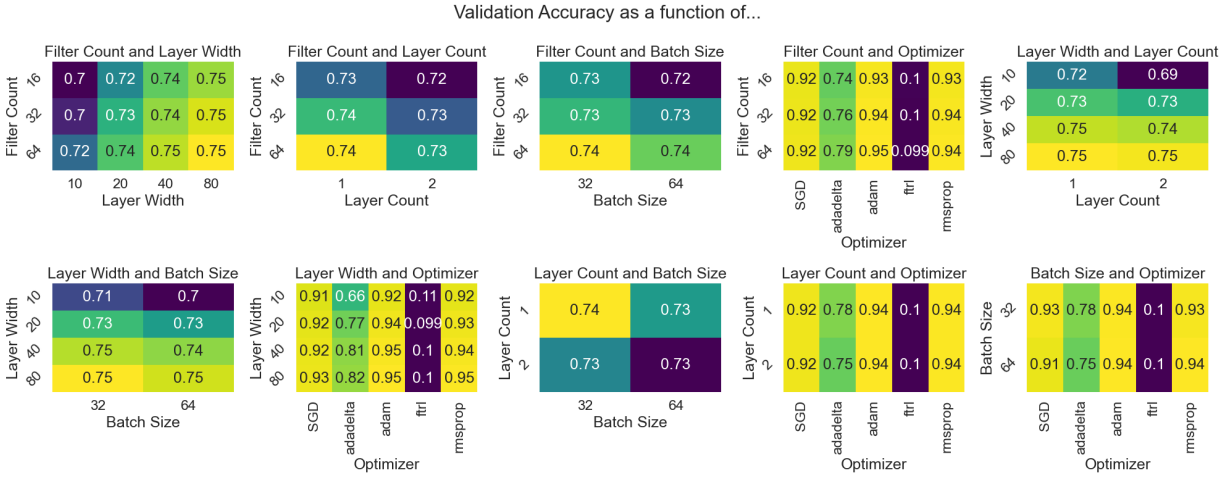

Fig. 4: Dataset 3 Hyperparameter Analysis based on Validation Accuracy

V. Value of Concepts Learned

With each classifier having their own unique challenges and architectures, all four members in their own way extracted value out of the concepts and techniques given by the overarching course. Most code used to generate these machine learning results consisted of pre-made libraries and functions, but having an understanding of the underlying mechanics made these algorithms more approachable.

Logistic Regression is a computationally efficient and simple algorithm that learns to predict a class based on probability. To avoid overfitting, L2 and/or L1 regularization can be used with tuned regularization strength and ratio of L1. This can reduce the effects of collinear and irrelevant features.

Support Vector Machine (SVM) is a supervised learning algorithm used for classification and regression. SVM tries to maximize the margin between the classes in the feature space, with the closes data points known as support vectors that define the margin. With the help of kernel functions like polynomial and RBF, it can classify the data that is non-linearly separable by mapping the features to higher dimensional space. The parameter C (regularization parameter) balances the trade-off between maximizing the margin and minimizing the classification errors. A Higher C value tries to classify the data correctly, but can lead to overfitting. While a lower C implies a wider margin and can underfit the model. The γ parameter decides the curvature/smoothness of the decision boundary. A low γ value can result in smooth decision boundary, while a high γ value can result in a complex decision boundary which may lead to overfit. Both parameters C and γ are crucial in tuning SVM for optimal performance.

PCA, as a powerful dimensionality reduction technique, allows us to transform a large set of variables into a smaller one that still contains most of the information in the large set. This is particularly valuable in dealing with high-dimensional data, where it helps to alleviate the curse of dimensionality and reduces computational cost. When we analyze datasets with the k-NN classifier, PCA not only expedites the computation but also potentially increases the accuracy by filtering out noise and irrelevant information. The k-NN algorithm's reliance on distance metrics to make predictions benefits significantly from the reduced-dimensional space provided by PCA.

Lastly, even a conceptual understanding of how neural networks function as a mathematical machine can enable one to leverage their strengths much more effectively. This allows for more informed control of the neural network's shape and complexity. On top of that, of course, it is very helpful to be aware of the many common practices and discoveries surrounding common neural networks, such as the fact that the ReLU, or Rectified Linear Unit function is an absolute staple in deciding on activation functions, or that the Adam and SGD optimizers are two of the most powerful and versatile optimizers in use.

VI. Conclusion

In the interests of exploring the behavior of varied parameters, we have developed a set of twelve classifier meant to handle three different classification tasks. Through grid search, we have found and selected for each classifier an optimal set of parameters based on their ability to accurately predict values of a carefully-made set of validation data, navigating computational limitations all the meanwhile. At last, these twelve classifiers were evaluated on their ability to make the same predictions on a larger set of test data, their performances appearing in Table VIII.

With the hyperparameters used, the Support Vector Machines classifier performs best in the two binary classification settings using the UCI datasets, while the Convolutional Neural Network does best on image analysis. It is worth mentioning, however, that most classifiers score very similarly on the same data set, perhaps owing to very rigorous hyperparameter searching.

Classifier	Logistic Regression	Support Vector Machines	k-Nearest Neighbors	Neural Networks
Test Accuracy: Dataset 1	97%	**98%**	95%	96.5%
Test Accuracy: Dataset 2	81%	**85%**	83%	81.5%
Test Accuracy: Dataset 3	84%	90%	86%	**91.1%**

TABLE VIII: Classifier Comparisons

Even with these respectable scores and suspected optimization of hyperparameters, there were some concessions made in the process of choosing hyperparameters. The neural network's high score in the Fashion MNIST dataset predictions appears as a success, but many aspects of its convolutional layer were unexplored simply for the sake of time.

Working with UCI Adult and FashionMNIST datasets also presented few challenges. Class imbalance is a significant issue in the Adult dataset, as it can lead to biased models and also the need to maintain data quality by handling missing values. Meanwhile, the FashionMNIST dataset is computationally intensive to work with due to its high dimensionality and generalizing the models to make multi classification add to its complexity. For all these datasets, and most real-world linearly inseparable data, logistic regression is not ideal as it has a linear decision boundary and cannot capture complex, non-linear relationships.

There are also other ways that challenges can manifest, less so in building a model, but in making a model worthwhile. Being one example, PCA and k-NN are extremely sensitive to initial variable variances, making it difficult to make a model of consistent quality. However, the constructive conclusions and insights gathered throughout the process of this study have shed much more light on the classifiers and datasets used so far, advancing our own understanding of Machine Learning.

VII. Appendix

A. Equations

Details related to SVM Kernels and its parameters

Kernel Type	**Parameters**	**Kernel Equation**	**Parameter Descriptions**	**Tuned Parameters**
Linear	C	$K(x, x') = x \cdot x'$	C: Regularization parameter	C
Polynomial	C, γ, r, d	$K(x, x') = (\gamma \cdot x \cdot x' + r)^d$	C: Regularization parameter, γ: Kernel coefficient r: Bias term (set to default value, 0 for hyperparameter tunning) d: Degree of the polynomial	C, γ, d
RBF (Radial Basis Function)	C, γ	$K(x, x') = \exp(-\gamma \cdot \|x - x'\|^2)$	C: Regularization parameter γ: Kernel coefficient	C, γ

TABLE IX: SVM Kernels and Their Parameters

B. Detailed Results of Logistic Regression Model for Dataset 1

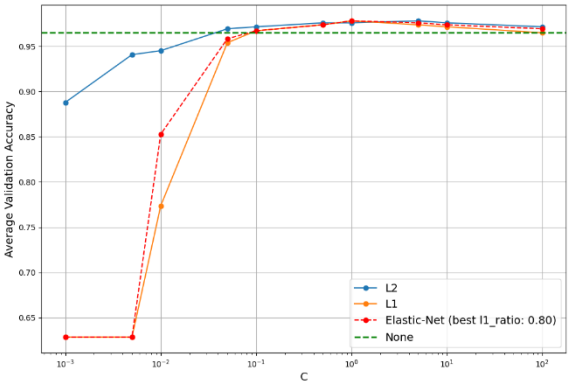

Fig. 5: Average Validation Accuracy vs C

Best parameters: {Elastic-Net, C: 1, L1 ratio: 0.8}

Confusion matrix:

$$\begin{pmatrix} 70 & 1 \\ 2 & 41 \end{pmatrix}$$

Classification report: Table X.

	Precision	Recall	F1-Score	Support
Benign	0.97	0.99	0.98	71
Malign	0.98	0.95	0.96	43
Accuracy	–	–	0.97	114
Macro Avg.	0.97	0.97	0.97	114
Weighted Avg.	0.97	0.97	0.97	114

TABLE X: Logistic Regression Result for Dataset 1

C. Detailed Results of SVM Model for Dataset 1

<table>
<tr><td></td><td colspan="5">Model with Tuned Parameters</td><td colspan="5">Baseline/Default Model</td></tr>
<tr><td>Parameters</td><td colspan="5">$kernel$: RBF
C: 5.08021804
γ: 0.017190722</td><td colspan="5">$kernel$: RBF
C: 1.0
γ: scale, $\frac{1}{n\_features \times \text{var}(X)}$</td></tr>
<tr><td rowspan="6">Scoring Metrics</td><td></td><td>Precision</td><td>Recall</td><td>F1-Score</td><td>Support</td><td></td><td>Precision</td><td>Recall</td><td>F1-Score</td><td>Support</td></tr>
<tr><td>Benign (Class: 0)</td><td>0.97</td><td>1.00</td><td>0.99</td><td>70</td><td>Benign (Class: 0)</td><td>0.96</td><td>1.00</td><td>0.98</td><td>70</td></tr>
<tr><td>Malignant (Class: 1)</td><td>1.00</td><td>0.95</td><td>0.98</td><td>44</td><td>Malignant (Class: 1)</td><td>1.00</td><td>0.93</td><td>0.96</td><td>44</td></tr>
<tr><td>Accuracy</td><td>–</td><td>–</td><td>0.98</td><td>114</td><td>Accuracy</td><td>–</td><td>–</td><td>0.97</td><td>114</td></tr>
<tr><td>Macro Avg.</td><td>0.99</td><td>0.98</td><td>0.98</td><td>114</td><td>Macro Avg.</td><td>0.98</td><td>0.97</td><td>0.97</td><td>114</td></tr>
<tr><td>Weighted Avg.</td><td>0.98</td><td>0.98</td><td>0.98</td><td>114</td><td>Weighted Avg.</td><td>0.97</td><td>0.97</td><td>0.97</td><td>114</td></tr>
<tr><td colspan="11">$BestAccuracy$: 98%</td></tr>
</table>

TABLE XI: SVM Model Evaluation based on Scoring Metrics for Dataset 1

Hyperparameter Tuning:
The search for parameter γ beyond 10^3 is not needed, as it can be observed from the validation curve (Fig 6) that the test accuracy decreases and leads to overfitting. [This is true for all kernels *(polynomial, RBF)*]

Similarly, the search for the parameter C beyond 10^6 is not needed, as it can be observed from the validation curve (Fig 7) that the test accuracy begins to decreases from $C \approx 10^2$. [This is true for all kernels *(linear, polynomial, RBF)*]

Kernel Type	Parameters and its values	Total Candidates Evaluated
Linear	C: 18 equidistant point are selected from the range $[10^{-3}, 10^{6}]$ on the log scale	18
Polynomial	C: 18 equidistant point are selected from the range $[10^{-3}, 10^{6}]$ on the log scale γ: 18 equidistant point are selected from the range $[10^{-6}, 10^{3}]$ on the log scale d: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]	$18 \times 18 \times 15 = 4860$
RBF (Radial Basis Function)	C: 18 equidistant point are selected from the range $[10^{-3}, 10^{6}]$ on the log scale γ: 18 equidistant point are selected from the range $[10^{-6}, 10^{3}]$ on the log scale	$18 \times 18 = 324$
Best Parameters for this Search - Kernel: RBF, C: 5.08021804, γ: 0.017190722		

TABLE XII: SVM Hyperparameter Tuning - I, for Dataset 1

Kernel Type	Parameters and its values	Total Candidates Evaluated
RBF (Radial Basis Function)	C: 1000 equidistant point are selected from the range [0.903, 5.082] on the log scale γ: 10 equidistant point are selected from the range [0.017, 1.719] on the log scale	$1000 \times 10 = 10000$
Best Parameters for this Search - Kernel: RBF, C: 5.08021804, γ: 0.017190722		

TABLE XIII: SVM Hyperparameter Tuning - II (refining the parameters with reduced search space)

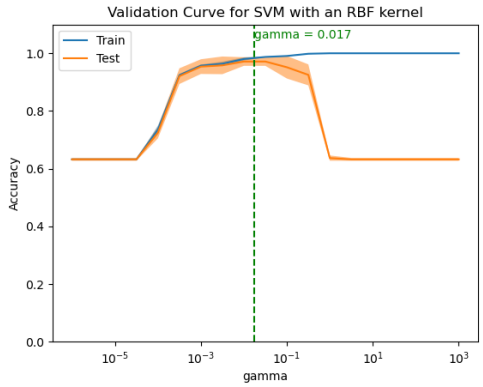

Fig. 6: Validation Curve for SVM with γ parameter

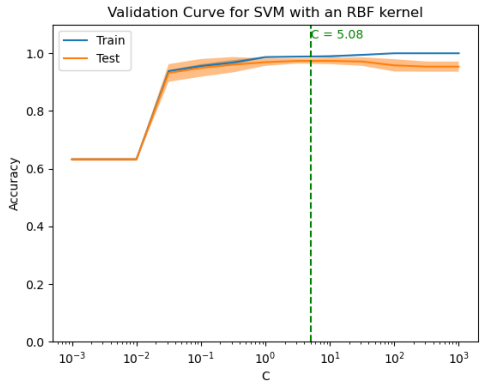

Fig. 7: Validation Curve for SVM with C parameter

D. Detailed Results of k-Nearest Neighbors Model for Dataset 1

Best parameters: {'n_neighbors': 3}
Confusion matrix:

$$\begin{pmatrix} 104 & 4 \\ 4 & 59 \end{pmatrix}$$

Training time: 0.56 seconds
Classification report: Table XIV

	Precision	Recall	F1-Score	Support
B	0.96	0.96	0.96	108
M	0.94	0.94	0.94	63
Accuracy	–	–	0.95	171
Macro Avg.	0.95	0.95	0.95	171
Weighted Avg.	0.95	0.95	0.95	171

TABLE XIV: k-NN Learning Result for Dataset 1

E. Detailed Results of Neural Network Model for Dataset 1

Tested with two 20-width hidden layers, batch size of 2, and 100 epochs on the adam optimizer.
Confusion matrix:

$$\begin{pmatrix} 73 & 3 \\ 2 & 36 \end{pmatrix}$$

Training time: 18.62 seconds
Classification report: Table XV

	Precision	Recall	F1-Score	Support
Benign	0.97	0.96	0.97	76
Malignant	0.92	0.95	0.94	38
Accuracy	–	–	0.96	114
Macro Avg.	0.95	0.95	0.95	114
Weighted Avg.	0.96	0.96	0.96	114

TABLE XV: Neural Network Learning Result for Dataset 1

F. Detailed Results of SVM Model for Dataset 2

Hyperparameter Tuning:

The search for parameter γ beyond 10^2 is not needed, as it can be observed from the validation curve (Fig 8) that the test accuracy decreases and leads to overfitting. [This is true for all kernels *(polynomial, RBF)*, and also going beyond $\gamma = 10^2$ can cause convergence issues for SVM model with *polynomial* kernel.]

Similarly, the search for the parameter C beyond 10^2 is not needed, as it can be observed from the validation curve (Fig 9) that the test accuracy begins to decreases from $C \approx 10^2$. [This is true for all kernels *(linear, polynomial, RBF)*, and also going beyond $C = 10^2$ can cause convergence issues for SVM model with *(polynomial, linear)* kernels.]

Kernel Type	**Parameters and its values**	**Total Candidates Evaluated**
Linear	C: 11 equidistant point are selected from the range $[10^{-3}, 10^2]$ on the log scale	11
Polynomial	C: 11 equidistant point are selected from the range $[10^{-3}, 10^2]$ on the log scale γ: 8 equidistant point are selected from the range $[10^{-5}, 10^2]$ on the log scale d: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	$11 \times 8 \times 10 = 880$
RBF (Radial Basis Function)	C: 11 equidistant point are selected from the range $[10^{-3}, 10^2]$ on the log scale γ: 17 equidistant point are selected from the range $[10^{-5}, 10^3]$ on the log scale	$11 \times 17 = 187$
Best Parameters for this Search - Kernel: RBF, C: 3.16227766, γ: 0.1		

TABLE XVI: SVM Hyperparameter Tuning - I, for Dataset 2

Kernel Type	**Parameters and its values**	**Total Candidates Evaluated**
RBF (Radial Basis Function)	C: 100 equidistant point are selected from the range [0.562, 3.162] on the log scale γ: 10 equidistant point are selected from the range [0.1, 10.0] on the log scale	$100 \times 10 = 1000$
Best Parameters for this Search - Kernel: RBF, C: 1.90679072, γ: 0.1		

TABLE XVII: SVM Hyperparameter Tuning - II (refining the parameters with reduced search space)

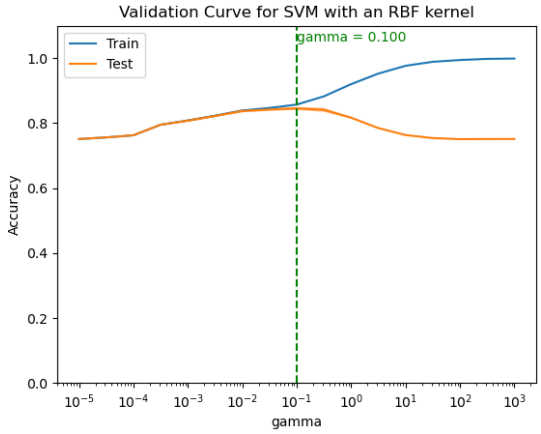

Fig. 8: Validation Curve for SVM with γ parameter

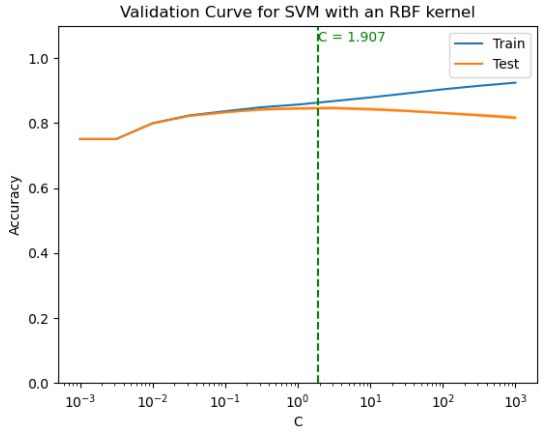

Fig. 9: Validation Curve for SVM with C parameter

G. Detailed Results of k-Nearest Neighbors Model for Dataset 2

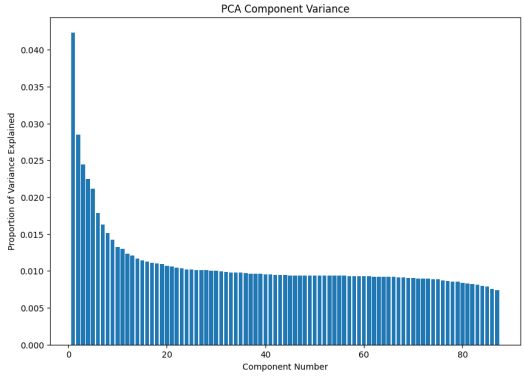

Fig. 10: PCA Component Variance for Dataset 2

Best parameters: {'k-NN_n-neighbors': 11}
Confusion matrix:

$$\begin{pmatrix} 9874 & 859 \\ 1585 & 1969 \end{pmatrix}$$

Training time: 96.29 seconds
Classification report: Table XVIII.

	Precision	Recall	F1-Score	Support
0	0.86	0.92	0.89	10733
1	0.70	0.55	0.62	3554
Accuracy	–	–	0.83	14287
Macro Avg.	0.78	0.74	0.75	14287
Weighted Avg.	0.82	0.83	0.82	14287

TABLE XVIII: k-NN Learning Result for Dataset 2

H. Detailed Results of Neural Network Model for Dataset 2

Tested with two 640-width hidden layers, batch size of 128, and 100 epochs on the adam optimizer.
Confusion matrix:

$$\begin{pmatrix} 6348 & 846 \\ 914 & 1417 \end{pmatrix}$$

Training time: 136.44 seconds
Classification report: Table XIX.

	Precision	Recall	F1-Score	Support
Below $50K	0.87	0.88	0.88	7194
Above $50K	0.63	0.61	0.62	2331
Accuracy	–	–	0.82	9525
Macro Avg.	0.75	0.75	0.75	9525
Weighted Avg.	0.81	0.82	0.81	9525

TABLE XIX: Neural Network Learning Result for Dataset 2

I. Detailed Results of Logistic Regression Model for Dataset 3

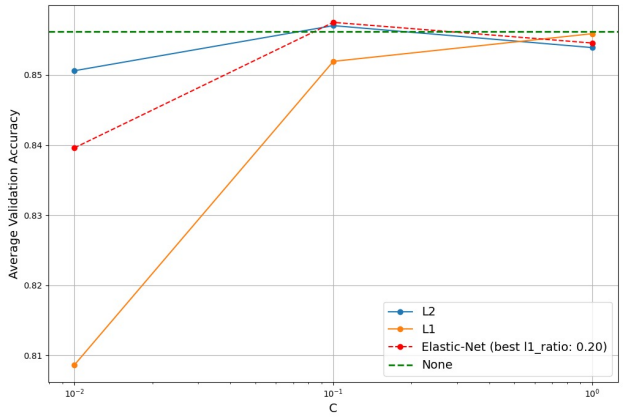

Fig. 11: LR: Average Accuracy VS C for Dataset 3

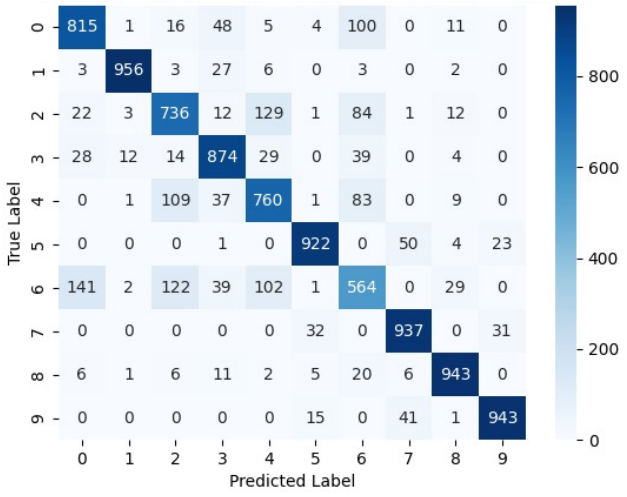

Fig. 12: Logistic Regression: Confusion Martrix for Dataset 3

J. Detailed Results of SVM Model for Dataset 3

Hyperparameter Tuning:

The search for parameter γ beyond 10^2 is not needed, as it can be observed from the validation curve (Fig 13) that the test accuracy decreases and leads to overfitting. [This is true for all kernels *(polynomial, RBF)*, and also going beyond $\gamma = 10^2$ can cause convergence issues for SVM model with *polynomial* kernel.]

Similarly, the search for the parameter C beyond 10^2 is not needed, as it can be observed from the validation curve (Fig 14) that the test accuracy begins to slightly decreases from $C \approx 10^2$. [This is true for all kernels *(linear, polynomial, RBF)*, and also going beyond $C = 10^2$ can cause convergence issues for SVM model with *(polynomial, linear)* kernels.]

Kernel Type	**Parameters and its values**	**Total Candidates Evaluated**
Linear	C: 7 equidistant point are selected from the range $[10^{-3}, 10^3]$ on the log scale	7
Polynomial	C: 7 equidistant point are selected from the range $[10^{-3}, 10^3]$ on the log scale γ: 7 equidistant point are selected from the range $[10^{-4}, 10^2]$ on the log scale d: [1, 2, 3, 4, 5]	$7 \times 7 \times 5 = 245$
RBF (Radial Basis Function)	C: 7 equidistant point are selected from the range $[10^{-3}, 10^3]$ on the log scale γ: 7 equidistant point are selected from the range $[10^{-4}, 10^2]$ on the log scale	$7 \times 7 = 49$
Best Parameters for this Search - Kernel: RBF, C: 10.0, γ: 0.1		

TABLE XX: SVM Hyperparameter Tuning - I, for Dataset 3

Kernel Type	**Parameters and its values**	**Total Candidates Evaluated**
RBF (Radial Basis Function)	C: 10 equidistant point are selected from the range [1.778, 100.0] on the log scale γ: 10 equidistant point are selected from the range [0.1, 1.0] on the log scale	$10 \times 10 = 100$
Best Parameters for this Search - Kernel: RBF, C: 4.35400465, γ: 0.166810053		

TABLE XXI: SVM Hyperparameter Tuning - II (refining the parameters with reduced search space)

Validation Curves for Parameters C and γ:

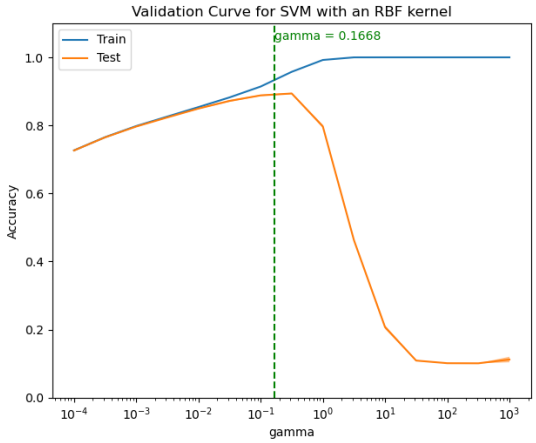

Fig. 13: Validation Curve for SVM with γ parameter

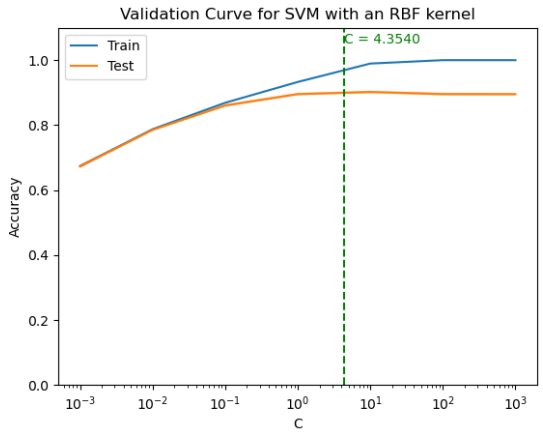

Fig. 14: Validation Curve for SVM with C parameter

Category	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Average**
Precision	0.85	0.99	0.82	0.90	0.83	0.98	0.74	0.95	0.98	0.97	0.90
Recall	0.85	0.97	0.84	0.90	0.84	0.97	0.71	0.97	0.98	0.96	0.90
F1-Score	0.85	0.98	0.83	0.90	0.83	0.98	0.72	0.96	0.98	0.97	0.90
Support	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	10000

TABLE XXII: SVM Model Classification Report for Dataset 3

K. Detailed Results of k-Nearest Neighbors Model for Dataset 3

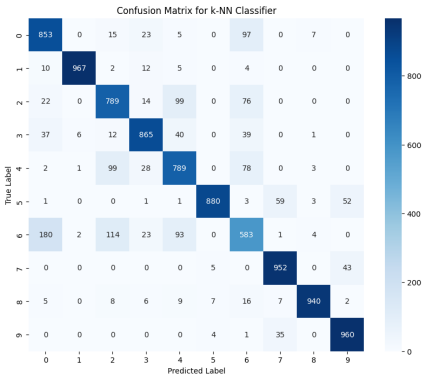

Fig. 15: k-NN Confusion Matrix for Dataset 3

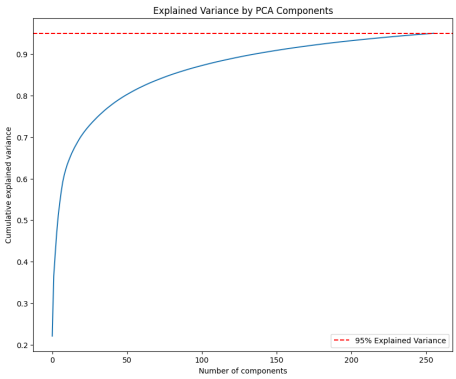

Fig. 16: PCA Component Variance for Dataset 3

Category	0	1	2	3	4	5	6	7	8	9	Average
Precision	0.77	0.99	0.76	0.89	0.76	0.98	0.65	0.90	0.98	0.91	0.86
Recall	0.85	0.97	0.79	0.86	0.79	0.88	0.58	0.95	0.94	0.96	0.86
F1-Score	0.81	0.98	0.77	0.88	0.77	0.93	0.61	0.93	0.96	0.93	0.86
Support	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	10000

TABLE XXIII: k-NN Learning Result for Dataset 3

L. Detailed Results of Neural Network Model for Dataset 3

Tested with 64 filter-convolutional layer, 80-width single hidden layer, batch size of 64, and 30 epochs on the adam optimizer. Training time was 202.93 seconds.

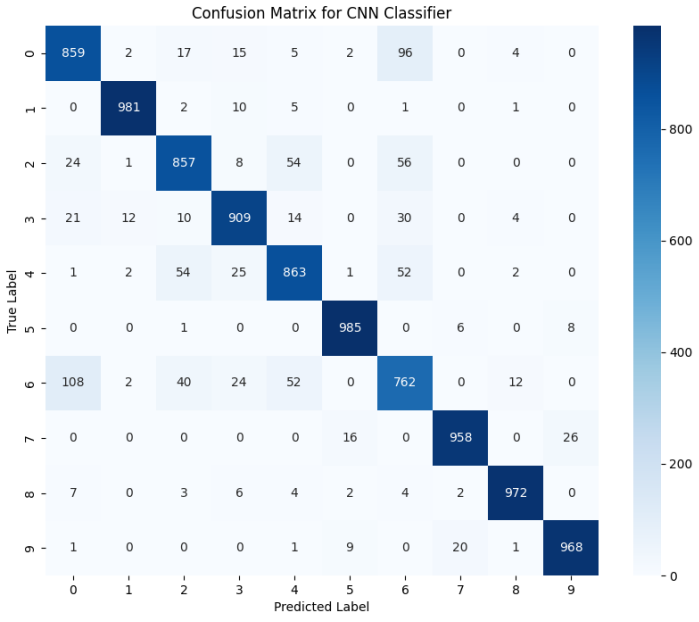

Fig. 17: Dataset 3 Neural Network Confusion Matrix

M. Code used for Dataset 1, Logistic Regression

```
## Code Execution Steps:
# - Simply run the logistic_regression.py file. It will print results for each dataset.
# - For a cleaner view, run the logistic_regression.ipynb file.
# - Here is dataset 1.
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV,cross_val_score
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns

def logreg_pipeline(X_train, X_test, y_train, y_test, dataset='Dataset'):
    # First Grid Search: L2 and L1
    param_grid_l2_l1 = {
        'C': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 100],
        'penalty': ['l2', 'l1']
    }

    # Create Logistic Regression model
    model = LogisticRegression(solver='saga', max_iter=10000)

    # Perform Grid Search for L2 and L1
    grid_search_l2_l1 = GridSearchCV(model, param_grid_l2_l1, cv=5, scoring='accuracy')
    grid_search_l2_l1.fit(X_train, y_train)

    # Second Grid Search: ElasticNet and l1_ratio
    param_grid_elasticnet = {
        'C': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 100],
        'penalty': ['elasticnet'],
        'l1_ratio': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
    }

    # Perform Grid Search for ElasticNet and l1_ratio
    grid_search_elasticnet = GridSearchCV(model, param_grid_elasticnet, cv=5,
        scoring='accuracy')
    grid_search_elasticnet.fit(X_train, y_train)

    # Extract the results
    results_l2_l1 = grid_search_l2_l1.cv_results_
    l2_means = results_l2_l1['mean_test_score'][results_l2_l1['param_penalty'] == 'l2']
    l1_means = results_l2_l1['mean_test_score'][results_l2_l1['param_penalty'] == 'l1']

    results_elasticnet = grid_search_elasticnet.cv_results_
    best_ratio = grid_search_elasticnet.best_params_['l1_ratio']
    elastic_means = results_elasticnet['mean_test_score'][results_elasticnet['param_l1_ratio']
        == best_ratio]

    # Logistic Regression with no regularization
    model_none = LogisticRegression(solver='saga', max_iter=10000, penalty=None)
    none_accuracies = cross_val_score(model_none, X_train, y_train, cv=5, scoring='accuracy')
    none_accuracy = np.mean(none_accuracies)

    # Plot the curves
    plt.figure(figsize=(12, 8))

    # L2 Average Accuracy Curve
    plt.plot(param_grid_l2_l1['C'], l2_means, label='L2', marker='o')

    # L1 Average Accuracy Curve
    plt.plot(param_grid_l2_l1['C'], l1_means, label='L1', marker='o')

    # Best ElasticNet l1_ratio Average Accuracy Curve
    plt.plot(param_grid_elasticnet['C'], elastic_means, label=f'Elastic-Net (best l1_ratio:
        {best_ratio:.2f})', linestyle='--', color='red', marker='o')

    # Logistic Regression with no regularization (constant line)
```

```
    plt.axhline(y=none_accuracy, color='green', linestyle='--', label='None', linewidth=2)

    # Set plot properties
    plt.title(f'{dataset} - Logistic Regression Grid Search')
    plt.xlabel('C',fontsize=14)
    plt.ylabel('Average Validation Accuracy',fontsize=14)
    plt.xscale('log')
    plt.legend(fontsize=14)
    plt.grid(True)
    plt.show()

    # Extract the best parameters and scores for each penalty
    best_params_l2_l1 = grid_search_l2_l1.best_params_
    best_score_l2_l1 = grid_search_l2_l1.best_score_

    best_params_elasticnet = grid_search_elasticnet.best_params_
    best_score_elasticnet = grid_search_elasticnet.best_score_

    # Choose the best parameters based on the highest mean test score
    best_params = max([(best_params_l2_l1, best_score_l2_l1),
                       (best_params_elasticnet, best_score_elasticnet),
                       ({'penalty':None},none_accuracy)],
                      key=lambda x: x[1])[0]

    print("Best Parameters L2/L1:", best_params_l2_l1)
    print("Best Score L2/L1:", best_score_l2_l1)
    print("Best Parameters ElasticNet:", best_params_elasticnet)
    print("Best Score ElasticNet:", best_score_elasticnet)
    print("Score None:", none_accuracy)
    print("Best Parameters Overall:", best_params)

    # Fit the best model with the full training set
    best_model = LogisticRegression(solver='saga', max_iter=10000, **best_params)
    best_model.fit(X_train, y_train)

    # Evaluate the best model on the test set
    y_pred = best_model.predict(X_test)

    # Display the classification report and confusion matrix
    print("Test Classification report:\n", classification_report(y_test, y_pred))
    print("Test Confusion Matrix:\n", confusion_matrix(y_test, y_pred))

    # Plot the confusion matrix
    confusion_mat = confusion_matrix(y_test, y_pred)
    sns.heatmap(confusion_mat, annot=True, fmt='d', cmap='Blues')
    plt.title(f'{dataset} - Logistic Regression Test Confusion Matrix')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

    return grid_search_l2_l1, grid_search_elasticnet

## Wisconsin Breast Cancer dataset
from ucimlrepo import fetch_ucirepo
from sklearn.preprocessing import LabelEncoder

# fetch dataset
breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)

# data (as pandas dataframes)
X = breast_cancer_wisconsin_diagnostic.data.features
y = breast_cancer_wisconsin_diagnostic.data.targets

# metadata
#print(breast_cancer_wisconsin_diagnostic.metadata)

# variable information
```

```
#print(breast_cancer_wisconsin_diagnostic.variables)

label_encoder = LabelEncoder()

X_fix = X.copy()
y_fix = y.copy()
y_fix['Diagnosis'] = label_encoder.fit_transform(y_fix['Diagnosis'])

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_fix, y_fix, test_size=0.2,
    random_state=42)

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

cancer_search_l2_l1, cancer_search_elasticnet = logreg_pipeline(X_train_scaled, X_test_scaled,
    y_train.to_numpy().flatten(), y_test, dataset='Cancer')

best_cancer_model = LogisticRegression(solver='saga', max_iter=10000,
    penalty='elasticnet',C=1,l1_ratio=0.8)
best_cancer_model.fit(X_train_scaled, y_train.to_numpy().flatten())
# Evaluate the best model on the test set
y_pred = best_cancer_model.predict(X_test_scaled)

# Display the classification report and confusion matrix
print("Test Classification report:\n", classification_report(y_test, y_pred))
print("Test Confusion Matrix:\n", confusion_matrix(y_test, y_pred))

# Plot the confusion matrix
confusion_mat = confusion_matrix(y_test, y_pred)
sns.heatmap(confusion_mat, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

N. Code used for Dataset 1, Support Vector Machines

```
## Code Execution Steps:
# - The dataset is already provided in "Data" directory, which is placed parallel to the code.
# - If the dataset is not present, download the dataset and place it in the "Data" directory.
# - Make sure that the relative path to the dataset is present
    "./data/breast+cancer+wisconsin+diagnostic/wdbc.data"
# - Run the python file. Ex: python ./breastCancer\_svm.py
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, accuracy_score, precision_score, recall_score,
    f1_score
import seaborn as sns
from sklearn.model_selection import ValidationCurveDisplay
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# read data from the breast+cancer+wisconsin+dignostic data set
    (./breast+cancer+wisconsin+dignostic)
wdbcDF = pd.read_csv('./data/breast+cancer+wisconsin+diagnostic/wdbc.data')

columnNames = [
    'ID',
    'Diagnosis',
    'radius1',
    'texture1',
    'perimeter1',
    'area1',
    'smoothness1',
    'compactness1',
    'concavity1',
    'concave_points1',
    'symmetry1',
    'fractal_dimension1 ',
    'radius2',
    'texture2',
    'perimeter2',
    'area2',
    'smoothness2',
    'compactness2',
    'concavity2',
    'concave_points2',
    'symmetry2',
    'fractal_dimension2',
    'radius3',
    'texture3',
    'perimeter3',
    'area3',
    'smoothness3',
    'compactness3',
    'concavity3',
    'concave_points3',
    'symmetry3',
    'fractal_dimension3'
]

wdbcDF.columns = columnNames

# drop rows with NaN
dropIndex = wdbcDF.isna().sum(axis=1).where(lambda x: x != 0).dropna().index
wdbcDF = wdbcDF.drop(dropIndex).reset_index().drop(columns='index')

```

```
# sort the ID in acsending order
wdbcDF = wdbcDF.sort_values(by='ID', ascending=True).reset_index().drop(columns='index')

# map the Diagosis values Malignant (M) = 1 and Benign (B) = 0
wdbcDF.loc[wdbcDF['Diagnosis'] == 'M', 'Diagnosis'] = 1
wdbcDF.loc[wdbcDF['Diagnosis'] == 'B', 'Diagnosis'] = 0
wdbcDF['Diagnosis'] = wdbcDF['Diagnosis'].astype(float)

# drop ID column
wdbcDF = wdbcDF.drop(columns='ID')

#seprate X (data) and Y (lable) from the data frame
data = wdbcDF.iloc[:,1:].to_numpy()
lable = wdbcDF.iloc[:,0].to_numpy()

#use Standard scaler to normalize the data
scalar = StandardScaler()
data = scalar.fit_transform(data)

#split the training and test data into 80% to 20% ratio respectively
xTrainData, xTestData, yTrainData, yTestData = train_test_split(data, lable, test_size=0.2,
    random_state=23)

# baseline/default SVM model
model = SVC(max_iter=10**8)

model.fit(xTrainData, yTrainData)

yPredict = model.predict(xTestData)

# Score report for the baseline/default SVM model
print(classification_report(yTestData,yPredict))

#Scoring Metrics
scoring = {'accuracy' : make_scorer(accuracy_score),
           'precision' : make_scorer(precision_score, zero_division=0),
           'recall' : make_scorer(recall_score, zero_division=0),
           'f1_score' : make_scorer(f1_score, zero_division=0)
}

# hyperparameter tuning and KFold cross validation
gammaValues = np.array([10**k for k in np.linspace(-6, 3, 18)])
gammaValues = gammaValues.tolist()
# Appending 'scale' and 'auto' to the list
gammaValues.append('scale')
gammaValues.append('auto')

cValues = [10**k for k in np.linspace(-3, 6, 18)]

parametersGridCV = [
    {
        'kernel' : ['linear'],
        'C' : cValues
    },
    {
        'kernel' : ['poly'],
        'C' : cValues,
        'degree' : list(range(1, 16, 1)),
        'gamma' : gammaValues
    },
    {
        'kernel' : ['rbf'],
        'C' : cValues,
        'gamma' : gammaValues
    }
]

```

```
gridCV = GridSearchCV(estimator=model, param_grid=parametersGridCV, cv=5, scoring=scoring,
    refit='accuracy', verbose=3, n_jobs=-1)

# fitting the model
gridCV.fit(xTrainData, yTrainData)
yPredict = gridCV.predict(xTestData)

# Display the best parameters
print(f'Best Parameters : {gridCV.best_params_}')

# Score report with best parameters after hyperparameter tunning for SVM model
print(classification_report(yTestData,yPredict))

# refine the C and gamma parameter
C = gridCV.best_params_['C']
gamma = gridCV.best_params_['gamma']
refineParametersGridCV = {
    'kernel' : [gridCV.best_params_['kernel']],
    'C' : [10**k for k in np.linspace(np.log10(C) - 0.75, np.log10(C) + 2, 1000)],
    'gamma' : [10**k for k in np.linspace(np.log10(gamma), np.log10(gamma) + 2, 10 )]
}

refineGridCV = GridSearchCV(estimator=model, param_grid=refineParametersGridCV, cv=5,
    scoring=scoring, refit='accuracy', verbose=3, n_jobs=-1)

# fitting the model
refineGridCV.fit(xTrainData, yTrainData)

# Display the best parameters
print(f'Best Parameters : {refineGridCV.best_params_}')

# predict on the test data
yPredict = refineGridCV.predict(xTestData)

# Score report with best parameters after hyperparameter tunning for SVM model
print(classification_report(yTestData,yPredict))

# Confusion Marix
confusionMatrix = confusion_matrix(yTestData, yPredict)
dispConfusionMatrix = ConfusionMatrixDisplay(confusion_matrix=confusionMatrix,
    display_labels=['Benign (Class = 0)', 'Malignant (Class = 1)'])
dispConfusionMatrix.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix for SVM Classifier")

# Display validation curves for training and test
gammaValue = 0.017190722018585746
validationCurve = ValidationCurveDisplay.from_estimator(
    SVC(kernel='rbf'),
    xTrainData,


    yTrainData,
    param_name="gamma",
    param_range=np.logspace(-6, 3, 19),
    score_type="both",
    n_jobs=2,
    score_name="Accuracy",
)
validationCurve.ax_.set_title("Validation Curve for SVM with an RBF kernel")
validationCurve.ax_.set_xlabel(r"gamma")
validationCurve.ax_.set_ylim(0.0, 1.1)
validationCurve.ax_.axvline(x=gammaValue, color='g', linestyle='--')
validationCurve.ax_.text(gammaValue, 1.05, 'gamma = {:.3f}'.format(gammaValue), color='g')
plt.show()

cValue = 5.080218046913023
validationCurve = ValidationCurveDisplay.from_estimator(
```

```
    SVC(kernel='rbf', gamma=0.01719),
    xTrainData,
    yTrainData,
    param_name="C",
    param_range=np.logspace(-3, 3, 13),
    score_type="both",
    n_jobs=2,
    score_name="Accuracy",
)
validationCurve.ax_.set_title("Validation Curve for SVM with an RBF kernel")
validationCurve.ax_.set_xlabel(r"C")
validationCurve.ax_.set_ylim(0.0, 1.1)
validationCurve.ax_.axvline(x=cValue, color='g', linestyle='--')
validationCurve.ax_.text(cValue, 1.05, 'C = {:.2f}'.format(cValue), color='g')
plt.show()
```

O. Code used for Dataset 1, k-Nearest Neighbors

```
## Code Execution Steps:
# - The %pip command will automatically install the dataset.
# - This code file was programmed via Google Colab. Change %pip to an appropriate syntax when running in environments other than that (i.e., pip or pip3).
# - Run the python file "PCA+k\_NN.py". Each section of code will provide analysis solution for dataset 1, 2 and 3. Here is dataset 1.
%pip install ucimlrepo
import numpy as np
import matplotlib.pyplot as plt
import torch, torchvision
import pandas as pd
import seaborn as sns
import time
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.compose import ColumnTransformer
from ucimlrepo import fetch_ucirepo

# Start the timmer
start_time = time.time()
breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)

# data (as pandas dataframes)
X = breast_cancer_wisconsin_diagnostic.data.features
y = breast_cancer_wisconsin_diagnostic.data.targets

# Step 1: Data Preprocessing
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 2: PCA for Dimensionality Reduction
pca = PCA(n_components=0.95)  # Choosing components that explain 95% of the variance
X_pca = pca.fit_transform(X_scaled)

# Step 3: k-NN Classifier Setup
knn = KNeighborsClassifier()

# Step 4: Cross-Validation and Model Evaluation
y_reshaped = y.values.ravel() if isinstance(y, pd.DataFrame) else y.ravel()
X_train, X_test, y_train, y_test = train_test_split(X_pca, y_reshaped, test_size=0.3, random_state=42)

# Hyperparameter Tuning
param_grid = {'n_neighbors': [3, 5, 7, 9, 11]}
grid = GridSearchCV(knn, param_grid, cv=5)
grid.fit(X_train, y_train)
end_time = time.time()
training_time = end_time - start_time

# Best parameters and model evaluation
print("Best parameters:", grid.best_params_)
print("Classification report:\n", classification_report(y_test, grid.predict(X_test)))
print("Confusion matrix:\n", confusion_matrix(y_test, grid.predict(X_test)))
print(f"Training time: {training_time:.2f} seconds")
sns.heatmap(confusion_matrix(y_test, grid.predict(X_test)), annot=True)
plt.title('Confusion Matrix for k-NN Classifier')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

P. Code used for Dataset 1, Neural Networks

```
## Code Execution Steps:
# - Run "pip3 install numpy matplotlib seaborn multiprocessing tensorflow scikit-learn
    ucimlrepo itertools pandas csv"
# - Modify Hyperparameter lists to explore different grid search arrangements.
# - Run python file using any Python >3.9 interpreter with the above libraries installed.
import numpy as np
import numpy.random as random
import matplotlib.pyplot as plt
import seaborn as sns
import multiprocessing
import os
import tensorflow as tf
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'  # Suppresses noisy outputs
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from ucimlrepo import fetch_ucirepo
import itertools
import pandas as pd
import csv
import time

def main():
    ## Hyperparameters
    PROPORTION_TRAIN = 0.8
    LAYER_DEPTHS = [10, 20, 40, 80, 160]
    HIDDEN_LAYERS = [1, 2]
    EPOCHS = 100
    BATCH_SIZES = [1, 2, 4, 8, 16, 32, 64]
    ACTIVATION_FUNCTIONS = "relu"
    LOSS_FUNCTION = "binary_crossentropy"
    OPTIMIZERS = ["SGD", "rmsprop", "adam", "adadelta", "ftrl"]

    filename = "FNN Results.csv"
    n_folds = 5
    kf = KFold(n_splits=n_folds)

    ## Data Pre-Processing
    #region
    breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)
    # data (as pandas dataframes)
    X = breast_cancer_wisconsin_diagnostic.data.features
    y = breast_cancer_wisconsin_diagnostic.data.targets

    # metadata
    # print(breast_cancer_wisconsin_diagnostic.metadata)
    # variable information
    # print(breast_cancer_wisconsin_diagnostic.variables)

    # Convert data to numpy
    X_combined = X.to_numpy()
    y_combined = y.to_numpy()

    # Labels are either 'M' or 'B', converting to 1 and 0.
    y_combined = np.where(y_combined == 'M', 1, 0)

    NUM = X_combined.shape[0]
    DIMS = X_combined.shape[1]

    # Shuffle data
    order_all = random.permutation(NUM)
    X_combined = X_combined[order_all,:]
    y_combined = y_combined[order_all,:]

    # Split data by train and test.
```

```
    splitIndex = int(PROPORTION_TRAIN * NUM)
    X_train = X_combined[:splitIndex,:] # Shape 455x30
    y_train = y_combined[:splitIndex,:] # Shape 455x1
    X_test = X_combined[splitIndex:,:] # Shape 114x30
    y_test = y_combined[splitIndex:,:] # Shape 114x1

    # Apply scaling to feature data, on the basis of the training dataset
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    #endregion

    #region
    # Prepare grid search over all combinations of hyperparameters
    num_param_states = len(LAYER_DEPTHS) * len(HIDDEN_LAYERS) * len(BATCH_SIZES) *
        len(OPTIMIZERS)
    combinations = list(itertools.product(LAYER_DEPTHS, HIDDEN_LAYERS, BATCH_SIZES,
        OPTIMIZERS))
    # Append indices to this large list
    combinations = [(x + (i,)) for i, x in enumerate(combinations)]

    # Prepare gigantic list of complete parameters.
    args_full = [(combo, DIMS, ACTIVATION_FUNCTIONS, LOSS_FUNCTION, X_train, y_train, EPOCHS,
        kf, n_folds) for combo in combinations]

    print(f"ready to iterate over {num_param_states} combinations")

    # Start running parallel trainings over all combinations
    outputData = []
    with multiprocessing.Pool() as pool:
        results = pool.starmap(eval_model, args_full)

        # Open the CSV file once and write all results
        with open(filename, 'w', newline='') as csvfile:
            writer = csv.writer(csvfile, delimiter=',')
            header = [["Index"], ["Depth"], ["Layers"], ["Batch Size"], ["Optimizer"], ["Valid
                Loss"], ["Valid Acc"], ["Time"]]
            writer.writerow(header)
            for result in results:
                outputData.append(result)
                writer.writerow(result)

    #endregion

    # Pick top 1 in terms of validation loss and get test error
    sorted_outputData = sorted(outputData, key=lambda x: x[5])
    top_1 = sorted_outputData[0]
    index, depth, hLayers, batchSize, optimizer, _, _, _ = top_1
    result = test_model((depth, hLayers, batchSize, optimizer, index), DIMS,
        ACTIVATION_FUNCTIONS, LOSS_FUNCTION, X_train, y_train, X_test, y_test, EPOCHS)
    print("Index, Depth, Layers, Batch Size, Optimizer, Test Loss, Test Accuracy, Training
        Time")
    print(result)

# Function to train and evaluate TF model
def eval_model(dynamic_args, d, activation, loss, X_train, y_train, epochs, kf, n_folds):
    # Unpack arguments
    depth, hLayers, batchSize, optimizer, index = dynamic_args
    # Clear previous models and graphs
    tf.keras.backend.clear_session()

    # Create a new graph for the current process
    with tf.Graph().as_default():
        # Create a session for the current graph
        with tf.compat.v1.Session() as sess:
            # Remind Tensorflow to keep quiet
```

```
            tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)

            # Model creation
            if hLayers == 1:
                model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])
            else:
                model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])

            # Compile the model
            model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

            # Start stopwatch
            start_time = time.time()

            # Initialize averaged results, test results are to be kept aside.
            valid_accuracy = 0
            valid_loss = 0
            # Train the model
            for index_train, index_val in kf.split(X_train):
                X_vtrain, X_val = X_train[index_train], X_train[index_val]
                y_vtrain, y_val = y_train[index_train], y_train[index_val]
                history = model.fit(X_vtrain, y_vtrain, epochs=epochs, batch_size=batchSize,
                    validation_split=0.2)
                # Test this model on the current fold's validation dataset, and the test set
                this_valid_loss, this_valid_accuracy = model.evaluate(X_val, y_val)

                # Add to average
                valid_accuracy += this_valid_accuracy / n_folds
                valid_loss += this_valid_loss / n_folds

            # Stop stopwatch
            end_time = time.time()


            training_time = end_time - start_time
            print(f"Done with Run {index}, t = {training_time}")
            return index, depth, hLayers, batchSize, optimizer, valid_loss, valid_accuracy,
                training_time

def test_model(dynamic_args, d, activation, loss, X_train, y_train, X_test, y_test, epochs):
    # Unpack arguments
    depth, hLayers, batchSize, optimizer, index = dynamic_args
    # Clear previous models and graphs
    tf.keras.backend.clear_session()

    # Create a new graph for the current process
    with tf.Graph().as_default():
        # Create a session for the current graph
        with tf.compat.v1.Session() as sess:
            # Remind Tensorflow to keep quiet
            tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)

            # Model creation
            if hLayers == 1:
                model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])
            else:
                model = tf.keras.models.Sequential([
```

```
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])

            # Compile the model
            model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

            # Start stopwatch
            start_time = time.time()

            # Train the model
            history = model.fit(X_train, y_train, epochs=epochs, batch_size = batchSize,
                validation_split=0.2)
            test_loss, test_accuracy = model.evaluate(X_test, y_test)
            # Stop stopwatch
            end_time = time.time()

            training_time = end_time - start_time
            print(f"Done with test, t = {training_time}")
            return index, depth, hLayers, batchSize, optimizer, test_loss, test_accuracy,
                training_time

if __name__ == "__main__":
    main()
```

```
## Code Execution Steps:
# - If you haven't already, run "pip3 install numpy matplotlib seaborn multiprocessing
    tensorflow scikit-learn ucimlrepo itertools pandas csv"
# - Modify individual hyperparameters to evaluate them on the test set.
# - Run python file using any Python >3.9 interpreter with the above libraries installed.
import numpy as np
import numpy.random as random
import os
import tensorflow as tf
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'  # Suppresses noisy outputs
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, classification_report
from ucimlrepo import fetch_ucirepo
import pandas as pd
import csv
import time

def main():
    ## Hyperparameters
    PROPORTION_TRAIN = 0.8
    layer_depth = 20
    hidden_layers = 2
    epochs = 100
    batch_size = 2
    activation_function = "relu"
    loss_function = "binary_crossentropy"
    optimizer = "adam"

    # filename = "FNNBC Evaluation Results.csv"

    ## Data Pre-Processing
    #region
    breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)
    # data (as pandas dataframes)
    X = breast_cancer_wisconsin_diagnostic.data.features
    y = breast_cancer_wisconsin_diagnostic.data.targets

    # metadata
    # print(breast_cancer_wisconsin_diagnostic.metadata)
```

```
    # variable information
    # print(breast_cancer_wisconsin_diagnostic.variables)

    # Convert data to numpy
    X_combined = X.to_numpy()
    y_combined = y.to_numpy()

    # Labels are either 'M' or 'B', converting to 1 and 0.
    y_combined = np.where(y_combined == 'M', 1, 0)

    NUM = X_combined.shape[0]
    DIMS = X_combined.shape[1]

    # Shuffle data
    order_all = random.permutation(NUM)
    X_combined = X_combined[order_all,:]
    y_combined = y_combined[order_all,:]

    # Split data by train and test.
    splitIndex = int(PROPORTION_TRAIN * NUM)
    X_train = X_combined[:splitIndex,:] # Shape 455x30
    y_train = y_combined[:splitIndex,:] # Shape 455x1
    X_test = X_combined[splitIndex:,:] # Shape 114x30
    y_test = y_combined[splitIndex:,:] # Shape 114x1

    # Apply scaling to feature data, on the basis of the training dataset
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    #endregion

    result = test_model((layer_depth, hidden_layers, batch_size, optimizer), DIMS,
        activation_function, loss_function, X_train, y_train, X_test, y_test, epochs)
    depth, hLayers, batchSize, optimizer, test_loss, test_accuracy, training_time, conmat,
        y_pred = result

    print("Depth, Layers, Batch Size, Optimizer, Test Loss, Test Accuracy, Training Time")
    print(f"{depth}, {hLayers}, {batchSize}, {optimizer}, {test_loss}, {test_accuracy},
        {training_time}")

    # Print confusion matrix
    print("Confusion Matrix:")
    print(conmat)

    # Compute and print classification report
    target_names = ['Class 0', 'Class 1']
    report = classification_report(y_test, (y_pred > 0.5), target_names=target_names)
    print("Classification Report:")
    print(report)


def test_model(dynamic_args, d, activation, loss, X_train, y_train, X_test, y_test, epochs):
    # Unpack arguments
    depth, hLayers, batchSize, optimizer = dynamic_args
    # Clear previous models and graphs
    tf.keras.backend.clear_session()

    # Create a new graph for the current process
    with tf.Graph().as_default():
        # Create a session for the current graph
        with tf.compat.v1.Session() as sess:
            # Remind Tensorflow to keep quiet
            tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)

            # Model creation
            if hLayers == 1:
```

```
                model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])
            else:
                model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])

            # Compile the model
            model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

            # Start stopwatch
            start_time = time.time()

            # Train the model
            history = model.fit(X_train, y_train, epochs=epochs, batch_size = batchSize,
                validation_split=0.2)
            test_loss, test_accuracy = model.evaluate(X_test, y_test)

            # Predict the test set
            y_pred = model.predict(X_test)

            # Compute confusion matrix
            conmat = confusion_matrix(y_test, (y_pred > 0.5))

            # Stop stopwatch
            end_time = time.time()

            training_time = end_time - start_time
            print(f"Done with test, t = {training_time}")

            return depth, hLayers, batchSize, optimizer, test_loss, test_accuracy,
                training_time, conmat, y_pred

if __name__ == "__main__":
    main()
```

Q. Code used for Dataset 2, Logistic Regression

```
## Code Execution Steps:
# - This is an extension of the Code from Dataset 1, as it uses the same imports and function.

## UCI Adult Dataset
from ucimlrepo import fetch_ucirepo

# fetch dataset
adult = fetch_ucirepo(id=2)

# data (as pandas dataframes)
X = adult.data.features
y = adult.data.targets

# metadata
#print(adult.metadata)

# variable information
#print(adult.variables)
X_copy = X.copy()
X_copy.replace('?', np.nan, inplace=True)
nan_indices = X_copy[X_copy.isnull().any(axis=1)].index
X_fix = X_copy.dropna()
y_fix = y.drop(nan_indices)
X_fix = X_fix.reset_index(drop=True)
y_fix = y_fix.reset_index(drop=True)

remove_periods = lambda x: x.replace('.', '') if isinstance(x, str) else x
X_fix = X_fix.applymap(remove_periods)
y_fix = y_fix.applymap(remove_periods)

# Initialize LabelEncoder
label_encoder = LabelEncoder()

# Identify categorical columns
categorical_columns = X_fix.select_dtypes(include=['object']).columns

# Encode categorical columns
for col in categorical_columns:
    X_fix[col] = label_encoder.fit_transform(X_fix[col])

y_fix['income'] = label_encoder.fit_transform(y_fix['income'])

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_fix, y_fix, test_size=0.2,
    random_state=42)
# Standardize the features using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

adult_search_l2_l1, adult_search_elasticnet = logreg_pipeline(X_train_scaled, X_test_scaled,
    y_train.to_numpy().flatten(), y_test, dataset='Adult')
```

R. Code used for Dataset 2, Support Vector Machines

```
## Code Execution Steps:
# - The dataset is already provided in "Data" directory, which is placed parallel to the code.
# - If the dataset is not present, download the dataset and place it in the "Data" directory.
# - Make sure that the relative path to the dataset is present "./data/adult/adult.data" and
    "./data/adult/adult.test"
# - Run the python file. Ex: python ./adult\_svm.py
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, accuracy_score, precision_score, recall_score,
    f1_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import ValidationCurveDisplay
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# column names
columnNames = [
    'age',
    'workclass',
    'fnlwgt',
    'education',
    'education-num',
    'marital-status',
    'occupation',
    'relationship',
    'race',
    'sex',
    'capital-gain',
    'capital-loss',
    'hours-per-week',
    'native-country',
    'income'
]

# read data from the Census Income data set (./adult.data , ./adult.test)
adultTrainDF = pd.read_csv('./data/adult/adult.data', names=columnNames)
adultTestDF = pd.read_csv('./data/adult/adult.test', names=columnNames)
adultTestDF = adultTestDF.drop(0)

# handle missing values in the data set
# Strip leading/trailing whitespace
adultTrainDF = adultTrainDF.applymap(lambda x: x.strip() if isinstance(x, str) else x)
adultTrainDF.replace('?', np.NaN, inplace=True)
adultTestnDF = adultTestDF.applymap(lambda x: x.strip() if isinstance(x, str) else x)
adultTestDF.replace(' ?', np.NaN, inplace=True)

# drop rows with NaN
dropIndex = adultTrainDF.isna().sum(axis=1).where(lambda x: x != 0).dropna().index
adultTrainDF = adultTrainDF.drop(dropIndex).reset_index().drop(columns='index')
dropIndex = adultTestDF.isna().sum(axis=1).where(lambda x: x != 0).dropna().index
adultTestDF = adultTestDF.drop(dropIndex).reset_index().drop(columns='index')

# map the adult income (>50K) = 1 and income (<=50K) = 0
adultTrainDF.loc[adultTrainDF['income'] == '>50K', 'income'] = 1
adultTrainDF.loc[adultTrainDF['income'] == '<=50K', 'income'] = 0
adultTrainDF['income'] = adultTrainDF['income'].astype(float)

adultTestDF.loc[adultTestDF['income'] == ' >50K.', 'income'] = 1
adultTestDF.loc[adultTestDF['income'] == ' <=50K.', 'income'] = 0
adultTestDF['income'] = adultTestDF['income'].astype(float)
```

```
64
65  # encode the categorical data
66  encodeColumnsList = ['workclass', 'education', 'marital-status', 'occupation', 'relationship',
        'race', 'sex', 'native-country']
67  encoder = LabelEncoder()
68  for column in encodeColumnsList:
69      adultTrainDF[column] = encoder.fit_transform(adultTrainDF[column])
70      adultTestDF[column] = encoder.fit_transform(adultTestDF[column])
71
72  #seprate X (data) and Y (lable) from the data frame
73  xTrainData = adultTrainDF.iloc[:,:-1].to_numpy()
74  yTrainData = adultTrainDF.iloc[:,-1].to_numpy()
75  xTestData = adultTestDF.iloc[:,:-1].to_numpy()
76  yTestData = adultTestDF.iloc[:,-1].to_numpy()
77
78  #use Standard scaler to normalize the data
79  scalar = StandardScaler()
80  xTrainData = scalar.fit_transform(xTrainData)
81  xTestData = scalar.fit_transform(xTestData)
82
83  # baseline/default SVM model
84  model = SVC(max_iter=10**7)
85
86  model.fit(xTrainData, yTrainData)
87
88  yPredict = model.predict(xTestData)
89
90  # Score report for the baseline/default SVM model
91  print(classification_report(yTestData,yPredict))
92
93  #Scoring Metrics
94  scoring = {'accuracy' : make_scorer(accuracy_score),
95             'precision' : make_scorer(precision_score, zero_division=0),
96             'recall' : make_scorer(recall_score, zero_division=0),
97             'f1_score' : make_scorer(f1_score, zero_division=0)
98  }
99
100 # hyperparameter tuning and KFold cross validation
101 gammaValues = np.array([10**k for k in np.linspace(-5, 3, 17)])
102 gammaValues = gammaValues.tolist()
103 # Appending 'scale' and 'auto' to the list
104 gammaValues.append('scale')
105 gammaValues.append('auto')
106
107 polyGammaValues1 = np.array([10**k for k in np.linspace(-5, 2, 8)])
108 polyGammaValues1 = polyGammaValues1.tolist()
109 polyGammaValues1.append('scale')
110 polyGammaValues1.append('auto')
111
112 polyGammaValues2 = np.array([10**k for k in np.linspace(-5, 2, 8)])
113 polyGammaValues2 = polyGammaValues2.tolist()
114 polyGammaValues2.append('scale')
115 polyGammaValues2.append('auto')
116
117 cValues = [10**k for k in np.linspace(-3, 2, 11)]
118
119 parametersGridCV = [
120     {
121         'kernel' : ['linear'],
122         'C' : cValues
123     },
124     {
125         'kernel' : ['poly'],
126         'C' : cValues,
127         'degree' : list(range(1, 5, 1)),
128         'gamma' : polyGammaValues2
129     },
```

```
    {
        'kernel' : ['poly'],
        'C' : cValues,
        'degree' : list(range(5, 11, 1)),
        'gamma' : polyGammaValues2
    },
    {
        'kernel' : ['rbf'],
        'C' : cValues,
        'gamma' : gammaValues
    }
]

gridCV = GridSearchCV(estimator=model, param_grid=parametersGridCV, cv=5, scoring=scoring,
    refit='accuracy', verbose=3, n_jobs=-1)

# fitting the model
gridCV.fit(xTrainData, yTrainData)
yPredict = gridCV.predict(xTestData)

# Display the best parameters
print(f'Best Parameters : {gridCV.best_params_}')

# Score report with best parameters after hyperparameter tunning for SVM model
print(classification_report(yTestData,yPredict))

# refine the C and gamma parameter
C = gridCV.best_params_['C']
gamma = gridCV.best_params_['gamma']
refineParametersGridCV = {
    'kernel' : [gridCV.best_params_['kernel']],
    'C' : [10**k for k in np.linspace(np.log10(C) - 0.75, np.log10(C) + 1, 100)],
    'gamma' : [10**k for k in np.linspace(np.log10(gamma), np.log10(gamma) + 2, 10 )]
}

refineGridCV = GridSearchCV(estimator=model, param_grid=refineParametersGridCV, cv=5,
    scoring=scoring, refit='accuracy', verbose=3, n_jobs=-1)

# fitting the model
refineGridCV.fit(xTrainData, yTrainData)

# Display the best parameters
print(f'Best Parameters : {refineGridCV.best_params_}')

# predict on the test data
yPredict = refineGridCV.predict(xTestData)

# Score report with best parameters after hyperparameter tunning for SVM model
print(classification_report(yTestData,yPredict))

# Confusion Marix
confusionMatrix = confusion_matrix(yTestData, yPredict)
dispConfusionMatrix = ConfusionMatrixDisplay(confusion_matrix=confusionMatrix,
    display_labels=['Income (<=50K): (Class = 0)', 'Income (>50K): (Class = 1)'])
dispConfusionMatrix.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix for SVM Classifier on Dataset 2")

# Display validation curves for training and test
gammaValue = 0.1
validationCurve = ValidationCurveDisplay.from_estimator(
    SVC(kernel='rbf'),
    xTrainData,
    yTrainData,
    param_name="gamma",
    param_range=np.logspace(-5, 3, 17),
    score_type="both",
    n_jobs=-1,
```

```
    score_name="Accuracy",
)
validationCurve.ax_.set_title("Validation Curve for SVM with an RBF kernel")
validationCurve.ax_.set_xlabel(r"gamma")
validationCurve.ax_.set_ylim(0.0, 1.1)
validationCurve.ax_.axvline(x=gammaValue, color='g', linestyle='--')
validationCurve.ax_.text(gammaValue, 1.05, 'gamma = {:.3f}'.format(gammaValue), color='g')
plt.show()

cValue = 1.906790722960592
validationCurve = ValidationCurveDisplay.from_estimator(
    SVC(kernel='rbf', gamma=0.1),
    xTrainData,
    yTrainData,
    param_name="C",
    param_range=np.logspace(-3, 3, 13),
    score_type="both",
    n_jobs=-1,
    score_name="Accuracy",
)
validationCurve.ax_.set_title("Validation Curve for SVM with an RBF kernel")
validationCurve.ax_.set_xlabel(r"C")
validationCurve.ax_.set_ylim(0.0, 1.1)
validationCurve.ax_.axvline(x=cValue, color='g', linestyle='--')
validationCurve.ax_.text(cValue, 1.05, 'C = {:.3f}'.format(cValue), color='g')
plt.show()
```

S. Code used for Dataset 2, k-Nearest Neighbors

```
## Code Execution Steps:
# - If you already changed %pip to an appropriate syntax in previous section, then just click RUN.
# - Run the python file "PCA+k\_NN.py". Each section of code will provide analysis solution for dataset 1, 2 and 3. Here is dataset 2.
%pip install ucimlrepo
import numpy as np
import matplotlib.pyplot as plt
import torch, torchvision
import pandas as pd
import seaborn as sns
import time
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.compose import ColumnTransformer
from ucimlrepo import fetch_ucirepo

# Start the timer
start_time = time.time()

# fetch dataset
adult = fetch_ucirepo(id=2)

# data (as pandas dataframes)
X = adult.data.features
y = adult.data.targets

# Handle missing values, 'None' is used for missing values, replace them with NaN
X.replace('None', np.nan, inplace=True)

# Drop rows with any NaN values
X.dropna(inplace=True)
y = y.loc[X.index]  # Align y with the rows kept in X

# Since we've removed rows, the indices may need to be reset
X.reset_index(drop=True, inplace=True)
y.reset_index(drop=True, inplace=True)

# Remove periods from the target variable
remove_periods = lambda x: x.replace('.', '') if isinstance(x, str) else x
y = y.applymap(remove_periods)

# One-hot encode the categorical variables
categorical_features = X.select_dtypes(include=['object']).columns
one_hot_encoder = OneHotEncoder(handle_unknown='ignore')
X_encoded = one_hot_encoder.fit_transform(X[categorical_features]).toarray()

# Standardize the numerical variables
numerical_features = X.select_dtypes(include=['float64', 'int64']).columns
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X[numerical_features])

# Combine the scaled and encoded features
X_processed = np.hstack((X_scaled, X_encoded))

# Encode the target variable
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y.squeeze())

# Split the dataset into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X_processed, y_encoded, test_size=0.3,
    random_state=42)

# Create the PCA + k-NN pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=0.95)),
    ('knn', KNeighborsClassifier())
])

# Hyperparameter tuning
param_grid = {
    'knn__n_neighbors': [3, 5, 7, 9, 11]
}
grid = GridSearchCV(pipeline, param_grid, cv=5)
grid.fit(X_train, y_train)
end_time = time.time()
training_time = end_time - start_time

# Best parameters and model evaluation
print("Best parameters:", grid.best_params_)
print("Classification report:\n", classification_report(y_test, grid.predict(X_test)))
print("Confusion matrix:\n", confusion_matrix(y_test, grid.predict(X_test)))
print(f"Training time: {training_time:.2f} seconds")

plt.figure(figsize=(10, 7))
confusion_mat = confusion_matrix(y_test, grid.predict(X_test))
sns.heatmap(confusion_mat, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix for k-NN Classifier')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

# Extract the PCA from the pipeline
pca = grid.best_estimator_.named_steps['pca']

# Plot the explained variance for each PCA component
plt.figure(figsize=(10, 7))
plt.bar(range(1, len(pca.explained_variance_ratio_) + 1), pca.explained_variance_ratio_)
plt.xlabel('Component Number')
plt.ylabel('Proportion of Variance Explained')
plt.title('PCA Component Variance')
plt.show()
```

T. Code used for Dataset 2, Neural Networks

```
## Code Execution Steps:
# - Run "pip3 install numpy matplotlib seaborn multiprocessing tensorflow scikit-learn
    ucimlrepo itertools pandas csv"
# - Modify Hyperparameter lists to explore different grid search arrangements.
# - Run python file using any Python >3.9 interpreter with the above libraries installed.
import numpy as np
import numpy.random as random
import matplotlib.pyplot as plt
import seaborn as sns
import torch, torchvision
import multiprocessing
import os
import tensorflow as tf
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'  # Suppresses noisy outputs
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.model_selection import KFold
from ucimlrepo import fetch_ucirepo
import itertools
import pandas as pd
import csv
import time

def main():
    ## Hyperparameters
    PROPORTION_TRAIN = 0.8
    LAYER_DEPTHS = [10, 20, 40, 80, 160, 320, 640]
    HIDDEN_LAYERS = [1, 2]
    EPOCHS = 100
    BATCH_SIZES = [16, 32, 64, 128]
    ACTIVATION_FUNCTIONS = "relu"
    LOSS_FUNCTION = "binary_crossentropy"
    OPTIMIZERS = ["SGD", "rmsprop", "adam", "adadelta", "ftrl"]

    filename = "FNN Results UCI.csv"
    n_folds = 5
    kf = KFold(n_splits=n_folds)

    ## Data Pre-Processing
    #region
    adult = fetch_ucirepo(id=2)
    # data (as pandas dataframes)
    X = adult.data.features
    y = adult.data.targets

    # metadata
    # print(adult.metadata)
    # variable information
    # print(adult.variables)

    # Get rid of NaNs
    combined = pd.concat([X, y], axis=1)
    combined = combined.dropna()
    X = combined.iloc[:,:-1]
    y = combined.iloc[:,-1]

    NUM = X.shape[0]

    # Shuffle Data
    combined_shuffled = combined.sample(frac=1).reset_index(drop=True)
    X_shuffled = combined_shuffled.iloc[:, :-1]
    y_shuffled = combined_shuffled.iloc[:, -1]

```

```
    # Split data by train and test.
    splitIndex = int(PROPORTION_TRAIN * NUM)
    X_train_pre = X_shuffled.iloc[:splitIndex, :]   # Shape 38096x14 (108 for onehot)
    y_train_pre = y_shuffled.iloc[:splitIndex]      # Shape 38096
    X_test_pre = X_shuffled.iloc[splitIndex:, :]    # Shape 9525x14 (108 for onehot)
    y_test_pre = y_shuffled.iloc[splitIndex:]       # Shape 9525

    # Convert categorical features to numerical
    categorical_cols = X.select_dtypes(include=['object', 'category']).columns
    numerical_cols = X.select_dtypes(include=['number']).columns
    # Preprocessing for numerical data
    numerical_transformer = StandardScaler()

    # Preprocessing for categorical data
    categorical_transformer = OneHotEncoder(handle_unknown='ignore')

    # Bundle preprocessing for numerical and categorical data
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numerical_transformer, numerical_cols),
            ('cat', categorical_transformer, categorical_cols)
        ])

    # Create a preprocessing and modeling pipeline
    clf = Pipeline(steps=[('preprocessor', preprocessor)])

    # Preprocess the features. Xs are in compressed sparse matrix format (sample, feature),
        value
    X_train = clf.fit_transform(X_train_pre)
    X_test = clf.transform(X_test_pre)

    # Labels are either '<=50K' or '>50K', converting to 1 and 0.
    y_train = y_train_pre.str.contains('>').astype(int)
    y_test = y_test_pre.str.contains('>').astype(int)

    DIMS = X_train.shape[1]

    #endregion

    #region
    # Prepare grid search over all combinations of hyperparameters
    num_param_states = len(LAYER_DEPTHS) * len(HIDDEN_LAYERS) * len(BATCH_SIZES) *
        len(OPTIMIZERS)
    combinations = list(itertools.product(LAYER_DEPTHS, HIDDEN_LAYERS, BATCH_SIZES,
        OPTIMIZERS))
    # Append indices to this large list
    combinations = [(x + (i,)) for i, x in enumerate(combinations)]

    # Prepare gigantic list of complete parameters.
    args_full = [(combo, DIMS, ACTIVATION_FUNCTIONS, LOSS_FUNCTION, X_train, y_train, EPOCHS,
        kf, n_folds) for combo in combinations]

    print(f"ready to iterate over {num_param_states} combinations")

    # Start running parallel trainings over all combinations
    outputData = []
    with multiprocessing.Pool() as pool:
        results = pool.starmap(eval_model, args_full)

        # Open the CSV file once and write all results
        with open(filename, 'w', newline='') as csvfile:
            writer = csv.writer(csvfile, delimiter=',')
            header = [["Index"], ["Depth"], ["Layers"], ["Batch Size"], ["Optimizer"], ["Valid
                Loss"], ["Valid Acc"], ["Time"]]
            writer.writerow(header)
            for result in results:
                outputData.append(result)
```

```
                writer.writerow(result)

    #endregion

    # Pick top 1 in terms of validation loss and get test error
    sorted_outputData = sorted(outputData, key=lambda x: x[5])
    top_1 = sorted_outputData[0]
    index, depth, hLayers, batchSize, optimizer, _, _, _ = top_1
    result = test_model((depth, hLayers, batchSize, optimizer, index), DIMS,
        ACTIVATION_FUNCTIONS, LOSS_FUNCTION, X_train, y_train, X_test, y_test, EPOCHS)
    print("Index, Depth, Layers, Batch Size, Optimizer, Test Loss, Test Accuracy, Training
        Time")
    print(result)

# Function to train and evaluate TF model
def eval_model(dynamic_args, d, activation, loss, X_train, y_train, epochs, kf, n_folds):
    # Unpack arguments
    depth, hLayers, batchSize, optimizer, index = dynamic_args
    # Clear previous models and graphs
    tf.keras.backend.clear_session()

    # Create a new graph for the current process
    with tf.Graph().as_default():
        # Create a session for the current graph
        with tf.compat.v1.Session() as sess:
            # Remind Tensorflow to keep quiet
            tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)

            # Model creation
            if hLayers == 1:
                model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])
            else:
                model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])

            # Compile the model
            model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

            # Start stopwatch
            start_time = time.time()

            # Initialize averaged results, test results are to be kept aside.
            valid_accuracy = 0
            valid_loss = 0
            # Train the model
            for index_train, index_val in kf.split(X_train):
                X_vtrain, X_val = X_train[index_train], X_train[index_val]
                y_vtrain, y_val = y_train[index_train], y_train[index_val]
                history = model.fit(X_vtrain, y_vtrain, epochs=epochs, batch_size=batchSize,
                    validation_split=0.2)
                # Test this model on the current fold's validation dataset, and the test set
                this_valid_loss, this_valid_accuracy = model.evaluate(X_val, y_val)

                # Add to average
                valid_accuracy += this_valid_accuracy / n_folds
                valid_loss += this_valid_loss / n_folds

            # Stop stopwatch
            end_time = time.time()
```

```
            training_time = end_time - start_time
            print(f"Done with Run {index}, t = {training_time}")
            return index, depth, hLayers, batchSize, optimizer, valid_loss, valid_accuracy,
                training_time

def test_model(dynamic_args, d, activation, loss, X_train, y_train, X_test, y_test, epochs):
    # Unpack arguments
    depth, hLayers, batchSize, optimizer, index = dynamic_args
    # Clear previous models and graphs
    tf.keras.backend.clear_session()

    # Create a new graph for the current process
    with tf.Graph().as_default():
        # Create a session for the current graph
        with tf.compat.v1.Session() as sess:
            # Remind Tensorflow to keep quiet
            tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)

            # Model creation
            if hLayers == 1:
                model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])
            else:
                model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])

            # Compile the model
            model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

            # Start stopwatch
            start_time = time.time()

            # Train the model
            history = model.fit(X_train, y_train, epochs=epochs, batch_size = batchSize,
                validation_split=0.2)
            test_loss, test_accuracy = model.evaluate(X_test, y_test)
            # Stop stopwatch
            end_time = time.time()

            training_time = end_time - start_time
            print(f"Done with test, t = {training_time}")
            return index, depth, hLayers, batchSize, optimizer, test_loss, test_accuracy,
                training_time

if __name__ == "__main__":
    main()
```

```
## Code Execution Steps:
# - If you haven't already, run "pip3 install numpy matplotlib seaborn multiprocessing
    tensorflow scikit-learn ucimlrepo itertools pandas csv"
# - Modify individual hyperparameters to evaluate them on the test set.
# - Run python file using any Python >3.9 interpreter with the above libraries installed.
import os
import tensorflow as tf
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'  # Suppresses noisy outputs
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import confusion_matrix, classification_report
from ucimlrepo import fetch_ucirepo
```

```
import itertools
import pandas as pd
import csv
import time

def main():
    ## Hyperparameters
    PROPORTION_TRAIN = 0.8
    layer_depth = 640
    hidden_layers = 2
    epochs = 100
    batch_size = 128
    activation_function = "relu"
    loss_function = "binary_crossentropy"
    optimizer = "adam"

    # filename = "FNN Results UCI.csv"

    ## Data Pre-Processing
    #region
    adult = fetch_ucirepo(id=2)
    # data (as pandas dataframes)
    X = adult.data.features
    y = adult.data.targets

    # metadata
    # print(adult.metadata)
    # variable information
    # print(adult.variables)

    # Get rid of NaNs
    combined = pd.concat([X, y], axis=1)
    combined = combined.dropna()
    X = combined.iloc[:,:-1]
    y = combined.iloc[:,-1]

    NUM = X.shape[0]

    # Shuffle Data
    combined_shuffled = combined.sample(frac=1).reset_index(drop=True)
    X_shuffled = combined_shuffled.iloc[:, :-1]
    y_shuffled = combined_shuffled.iloc[:, -1]

    # Split data by train and test.
    splitIndex = int(PROPORTION_TRAIN * NUM)
    X_train_pre = X_shuffled.iloc[:splitIndex, :]   # Shape 38096x14 (108 for onehot)
    y_train_pre = y_shuffled.iloc[:splitIndex]      # Shape 38096
    X_test_pre = X_shuffled.iloc[splitIndex:, :]    # Shape 9525x14 (108 for onehot)
    y_test_pre = y_shuffled.iloc[splitIndex:]       # Shape 9525

    # Convert categorical features to numerical
    categorical_cols = X.select_dtypes(include=['object', 'category']).columns
    numerical_cols = X.select_dtypes(include=['number']).columns
    # Preprocessing for numerical data
    numerical_transformer = StandardScaler()

    # Preprocessing for categorical data
    categorical_transformer = OneHotEncoder(handle_unknown='ignore')

    # Bundle preprocessing for numerical and categorical data
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numerical_transformer, numerical_cols),
            ('cat', categorical_transformer, categorical_cols)
        ])

    # Create a preprocessing and modeling pipeline
```

```
    clf = Pipeline(steps=[('preprocessor', preprocessor)])

    # Preprocess the features. Xs are in compressed sparse matrix format (sample, feature), value
    X_train = clf.fit_transform(X_train_pre)
    X_test = clf.transform(X_test_pre)

    # Labels are either '<=50K' or '>50K', converting to 1 and 0.
    y_train = y_train_pre.str.contains('>').astype(int)
    y_test = y_test_pre.str.contains('>').astype(int)

    DIMS = X_train.shape[1]

    #endregion

    result = test_model((layer_depth, hidden_layers, batch_size, optimizer), DIMS, activation_function, loss_function, X_train, y_train, X_test, y_test, epochs)
    depth, hLayers, batchSize, optimizer, test_loss, test_accuracy, training_time, conmat, y_pred = result

    print("Depth, Layers, Batch Size, Optimizer, Test Loss, Test Accuracy, Training Time")
    print(f"{depth}, {hLayers}, {batchSize}, {optimizer}, {test_loss}, {test_accuracy}, {training_time}")

    # Print confusion matrix
    print("Confusion Matrix:")
    print(conmat)

    # Compute and print classification report
    target_names = ['Class 0', 'Class 1']
    report = classification_report(y_test, (y_pred > 0.5), target_names=target_names)
    print("Classification Report:")
    print(report)


def test_model(dynamic_args, d, activation, loss, X_train, y_train, X_test, y_test, epochs):
    # Unpack arguments
    depth, hLayers, batchSize, optimizer = dynamic_args
    # Clear previous models and graphs
    tf.keras.backend.clear_session()

    # Create a new graph for the current process
    with tf.Graph().as_default():
        # Create a session for the current graph
        with tf.compat.v1.Session() as sess:
            # Remind Tensorflow to keep quiet
            tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)

            # Model creation
            if hLayers == 1:
                model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])
            else:
                model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(depth, activation=activation, input_shape=(d,)),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])

            # Compile the model
            model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

            # Start stopwatch
            start_time = time.time()

```

```
144             # Train the model
145             history = model.fit(X_train, y_train, epochs=epochs, batch_size = batchSize,
                    validation_split=0.2)
146             test_loss, test_accuracy = model.evaluate(X_test, y_test)
147 
148             # Predict the test set
149             y_pred = model.predict(X_test)
150 
151             # Compute confusion matrix
152             conmat = confusion_matrix(y_test, (y_pred > 0.5))
153 
154             # Stop stopwatch
155             end_time = time.time()
156 
157             training_time = end_time - start_time
158             print(f"Done with test, t = {training_time}")
159 
160             return depth, hLayers, batchSize, optimizer, test_loss, test_accuracy,
                    training_time, conmat, y_pred
161 
162 if __name__ == "__main__":
163     main()
```

U. Code used for Dataset 3, Logistic Regression

```
## Code Execution Steps:
# - This code can be run independently from the code from the other datasets.

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV,cross_val_score
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
def logreg_fashion_pipeline(X_train, X_test, y_train, y_test, dataset='Fashion'):
    # First Grid Search: L2 and L1
    param_grid_l2_l1 = {
        'C': [0.01, 0.1, 1],
        'penalty': ['l2', 'l1']
    }

    # Create Logistic Regression model
    model = LogisticRegression(solver='saga',max_iter=100)

    # Perform Grid Search for L2 and L1
    grid_search_l2_l1 = GridSearchCV(model, param_grid_l2_l1, cv=5, scoring='accuracy')
    grid_search_l2_l1.fit(X_train, y_train)
    print('finished L2, L1 grid search CV')

    # Second Grid Search: ElasticNet and l1_ratio
    param_grid_elasticnet = {
        'C': [0.01, 0.1, 1],
        'penalty': ['elasticnet'],
        'l1_ratio': [0.2, 0.4, 0.6, 0.8]
    }

    model = LogisticRegression(solver='saga',max_iter=100)

    # Perform Grid Search for ElasticNet and l1_ratio
    grid_search_elasticnet = GridSearchCV(model, param_grid_elasticnet, cv=5,
        scoring='accuracy')
    grid_search_elasticnet.fit(X_train, y_train)
    print('finished elastic-net grid search CV')

    # Extract the results
    results_l2_l1 = grid_search_l2_l1.cv_results_
    l2_means = results_l2_l1['mean_test_score'][results_l2_l1['param_penalty'] == 'l2']
    l1_means = results_l2_l1['mean_test_score'][results_l2_l1['param_penalty'] == 'l1']

    results_elasticnet = grid_search_elasticnet.cv_results_
    best_ratio = grid_search_elasticnet.best_params_['l1_ratio']
    elastic_means = results_elasticnet['mean_test_score'][results_elasticnet['param_l1_ratio']
        == best_ratio]

    # Logistic Regression with no regularization
    model_none = LogisticRegression(max_iter=100, penalty=None)
    none_accuracies = cross_val_score(model_none, X_train, y_train, cv=5, scoring='accuracy')
    print('finished None CV')
    none_accuracy = np.mean(none_accuracies)

    # Plot the curves
    plt.figure(figsize=(12, 8))

    # L2 Average Accuracy Curve
    plt.plot(param_grid_l2_l1['C'], l2_means, label='L2', marker='o')

    # L1 Average Accuracy Curve
    plt.plot(param_grid_l2_l1['C'], l1_means, label='L1', marker='o')

    # Best ElasticNet l1_ratio Average Accuracy Curve
```

```
    plt.plot(param_grid_elasticnet['C'], elastic_means, label=f'Elastic-Net (best l1_ratio:
        {best_ratio:.2f})', linestyle='--', color='red', marker='o')

    # Logistic Regression with no regularization (constant line)
    plt.axhline(y=none_accuracy, color='green', linestyle='--', label='None', linewidth=2)

    # Set plot properties
    plt.title(f'{dataset} - Logistic Regression Grid Search')
    plt.xlabel('C',fontsize=14)
    plt.ylabel('Average Validation Accuracy',fontsize=14)
    plt.xscale('log')
    plt.legend(fontsize=14)
    plt.grid(True)
    plt.show()

    # Extract the best parameters and scores for each penalty
    best_params_l2_l1 = grid_search_l2_l1.best_params_
    best_score_l2_l1 = grid_search_l2_l1.best_score_

    best_params_elasticnet = grid_search_elasticnet.best_params_
    best_score_elasticnet = grid_search_elasticnet.best_score_

    # Choose the best parameters based on the highest mean test score
    best_params = max([(best_params_l2_l1, best_score_l2_l1),
                       (best_params_elasticnet, best_score_elasticnet),
                       ({'penalty':None},none_accuracy)],
                      key=lambda x: x[1])[0]

    print("Best Parameters L2/L1:", best_params_l2_l1)
    print("Best Score L2/L1:", best_score_l2_l1)
    print("Best Parameters ElasticNet:", best_params_elasticnet)
    print("Best Score ElasticNet:", best_score_elasticnet)
    print("Score None:", none_accuracy)
    print("Best Parameters Overall:", best_params)

    # Fit the best model with the full training set
    best_model = LogisticRegression(solver='saga', max_iter=100, **best_params)
    best_model.fit(X_train, y_train)
    print('finished best model fit')

    # Evaluate the best model on the test set
    y_pred = best_model.predict(X_test)

    # Display the classification report and confusion matrix
    print("Test Classification report:\n", classification_report(y_test, y_pred))
    print("Test Confusion Matrix:\n", confusion_matrix(y_test, y_pred))

    # Plot the confusion matrix
    confusion_mat = confusion_matrix(y_test, y_pred)
    sns.heatmap(confusion_mat, annot=True, fmt='d', cmap='Blues')
    plt.title(f'{dataset} - Logistic Regression Test Confusion Matrix')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

    return grid_search_l2_l1, grid_search_elasticnet

## import Fashion MNIST data
import numpy as np
import torch, torchvision
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]))
```

```
129 X_test = X_test.reshape((X_test.shape[0], X_test.shape[1]*X_test.shape[2]))
130 X_train = X_train/255.0
131 X_test = X_test/255.0
```

V. Code used for Dataset 3, Support Vector Machines

```
## Code Execution Steps:
# - Run the python file. Ex: python ./fashionMNIST\_svm.py
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch, torchvision
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, accuracy_score, precision_score, recall_score,
    f1_score
from sklearn.model_selection import ValidationCurveDisplay
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay


# Load the Fashion MNIST Data
trainData = torchvision.datasets.FashionMNIST("./data", download=True)
testData = torchvision.datasets.FashionMNIST("./data", download=True, train=False)

# reshape the images from (28x28) to (14x14)
resize = torchvision.transforms.Resize((14, 14))
trainDataResize = resize(trainData.data)
testDataResize = resize(testData.data)

xTrainData = trainDataResize.data.numpy()
yTrainData = trainData.targets.numpy()
xTestData = testDataResize.data.numpy()
yTestData = testData.targets.numpy()

# Normalize the data
xTrainData = xTrainData.reshape((xTrainData.shape[0], xTrainData.shape[1]*xTrainData.shape[2]))
xTestData = xTestData.reshape((xTestData.shape[0], xTestData.shape[1]*xTestData.shape[2]))
xTrainData = xTrainData/255.0
xTestData = xTestData/255.0

# baseline/default SVM model
model = SVC(max_iter=10**5)

model.fit(xTrainData, yTrainData)

yPredict = model.predict(xTestData)

# Score report for the baseline/default SVM model
print(classification_report(yTestData,yPredict))

#Scoring Metrics
scoring = {'accuracy' : make_scorer(accuracy_score),
           'precision' : make_scorer(precision_score, average='macro', zero_division=0),
           'recall' : make_scorer(recall_score, average='macro', zero_division=0),
           'f1_score' : make_scorer(f1_score, average='macro', zero_division=0)
}

# hyperparameter tuning and KFold cross validation
gammaValues = np.array([10**k for k in np.linspace(-4, 2, 7)])
gammaValues = gammaValues.tolist()
# Appending 'scale' and 'auto' to the list
gammaValues.append('scale')
gammaValues.append('auto')

parametersGridCV = [
    {
        'kernel' : ['linear'],
        'C' : [10**k for k in np.linspace(-4, 1, 6)]
    },
```

```
    {
        'kernel' : ['poly'],
        'C' : [10**k for k in np.linspace(-3, 3, 7)],
        'degree' : list(range(1, 6, 1)),
        'gamma' : gammaValues
    },
    {
        'kernel' : ['rbf'],
        'C' : [10**k for k in np.linspace(-3, 3, 7)],
        'gamma' : gammaValues
    }
]

gridCV = GridSearchCV(estimator=model, param_grid=parametersGridCV, cv=5, scoring=scoring,
    refit='accuracy', verbose=3, n_jobs=-1)

# fitting the model
gridCV.fit(xTrainData, yTrainData)
yPredict = gridCV.predict(xTestData)

print(f'Best Parameters : {gridCV.best_params_}')

# Score report with best parameters after hyperparameter tunning for SVM model
print(classification_report(yTestData,yPredict))

# refine the C and gamma parameter
C = gridCV.best_params_['C']
gamma = gridCV.best_params_['gamma']
refineParametersGridCV = {
    'kernel' : [gridCV.best_params_['kernel']],
    'C' : [10**k for k in np.linspace(np.log10(C) - 0.75, np.log10(C) + 1, 10)],
    'gamma' : [10**k for k in np.linspace(np.log10(gamma), np.log10(gamma) + 1, 10 )]
}

refineGridCV = GridSearchCV(estimator=model, param_grid=refineParametersGridCV, cv=5,
    scoring=scoring, refit='accuracy', verbose=3, n_jobs=-1)

# fitting the model
refineGridCV.fit(xTrainData, yTrainData)

# Display the best parameters
print(f'Best Parameters : {refineGridCV.best_params_}')

# predict on the test data
yPredict = refineGridCV.predict(xTestData)

# Score report with best parameters after hyperparameter tunning for SVM model
print(classification_report(yTestData,yPredict))

# Confusion Marix
confusionMatrix = confusion_matrix(yTestData, yPredict)
dispConfusionMatrix = ConfusionMatrixDisplay(confusion_matrix=confusionMatrix)
dispConfusionMatrix.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix for SVM Classifier on FashionMNIST Dataset")

# Display validation curves for training and test
gammaValue = 0.16681005372000587
validationCurve = ValidationCurveDisplay.from_estimator(
    SVC(kernel='rbf'),
    xTrainData,
    yTrainData,
    param_name="gamma",
    param_range=np.logspace(-4, 3, 8),
    score_type="both",
    n_jobs=-1,
    score_name="Accuracy",
)
```

```
validationCurve.ax_.set_title("Validation Curve for SVM with an RBF kernel")
validationCurve.ax_.set_xlabel(r"gamma")
validationCurve.ax_.set_ylim(0.0, 1.1)
validationCurve.ax_.axvline(x=gammaValue, color='g', linestyle='--')
validationCurve.ax_.text(gammaValue, 1.05, 'gamma = {:.4f}'.format(gammaValue), color='g')
plt.show()

cValue = 4.354004653656649
validationCurve = ValidationCurveDisplay.from_estimator(
    SVC(kernel='rbf', gamma=0.16681005372000587),
    xTrainData,
    yTrainData,
    param_name="C",
    param_range=np.logspace(-3, 3, 7),
    score_type="both",
    n_jobs=-1,
    score_name="Accuracy",
)
validationCurve.ax_.set_title("Validation Curve for SVM with an RBF kernel")
validationCurve.ax_.set_xlabel(r"C")
validationCurve.ax_.set_ylim(0.0, 1.1)
validationCurve.ax_.axvline(x=cValue, color='g', linestyle='--')
validationCurve.ax_.text(cValue, 1.05, 'C = {:.4f}'.format(cValue), color='g')
plt.show()
```

W. Code used for Dataset 3, k-Nearest Neighbors

```
## Code Execution Steps:
# - If you already changed %pip to an appropriate syntax in previous section, then just click RUN.
# - Run the python file "PCA+k\_NN.py". Each section of code will provide analysis solution for dataset 1, 2 and 3. Here is dataset 3.
%pip install ucimlrepo
import numpy as np
import matplotlib.pyplot as plt
import torch, torchvision
import pandas as pd
import seaborn as sns
import time
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.compose import ColumnTransformer
from ucimlrepo import fetch_ucirepo

# Start the timmer
start_time = time.time()

# Load Fashion MNIST dataset
train_set = torchvision.datasets.FashionMNIST("./data", download = True)
test_set = torchvision.datasets.FashionMNIST("./data", download = True, train = False)

# Prepare training and testing data
X_train = train_set.data.numpy().reshape((train_set.data.size(0), -1)) / 255.0
labels_train = train_set.targets.numpy()
X_test = test_set.data.numpy().reshape((test_set.data.size(0), -1)) / 255.0
labels_test = test_set.targets.numpy()

# Create the PCA + k-NN pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=0.95)),  # Retain 95% of the variance
    ('knn', KNeighborsClassifier(n_neighbors=5))  # Start with k=5
])

# Fit the pipeline to the training data
pipeline.fit(X_train, labels_train)

# Predict the labels for the test set
labels_pred = pipeline.predict(X_test)

# Classification report
print(classification_report(labels_test, labels_pred))

# Confusion matrix
conf_mat = confusion_matrix(labels_test, labels_pred)

# Plot the confusion matrix
print(f"Training time: {training_time:.2f} seconds")
plt.figure(figsize=(10, 8))
sns.heatmap(conf_mat, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix for k-NN Classifier')
plt.show()

# Extract the PCA from the pipeline
pca = pipeline.named_steps['pca']
```

```
end_time = time.time()
training_time = end_time - start_time

# Plot the cumulative explained variance to decide on the number of components to retain
plt.figure(figsize=(10, 8))
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('Number of components')
plt.ylabel('Cumulative explained variance')
plt.title('Explained Variance by PCA Components')
plt.axhline(y=0.95, color='r', linestyle='--', label='95% Explained Variance')
plt.legend(loc='best')
plt.show()
```

X. Code used for Dataset 3, Neural Networks

```
## Code Execution Steps:
# - Run "pip3 install numpy matplotlib seaborn torch torchvision multiprocessing tensorflow
    scikit-learn ucimlrepo itertools pandas csv"
# - Modify Hyperparameter lists to explore different grid search arrangements.
# - Run python file using any Python >3.9 interpreter with the above libraries installed.
import numpy as np
import numpy.random as random
import matplotlib.pyplot as plt
import seaborn as sns
import torch, torchvision
import multiprocessing
import os
import tensorflow as tf
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'  # Suppresses noisy outputs
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
import itertools
import pandas as pd
import csv
import time

def main():
    ## Hyperparameters
    FILTERS = [16, 32, 64]
    LAYER_DEPTHS = [10, 20, 40, 80]
    HIDDEN_LAYERS = [1, 2]
    EPOCHS = 30
    BATCH_SIZES = [32, 64]
    ACTIVATION_FUNCTIONS = "relu"
    LOSS_FUNCTION = "sparse_categorical_crossentropy"
    OPTIMIZERS = ["SGD", "rmsprop", "adam", "adadelta", "ftrl"]

    filename = "CNN Results.csv"
    n_folds = 5
    kf = KFold(n_splits=n_folds)

    ## Data Pre-Processing
    #region
    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()
    # Add a fourth dimension to satisfy conv2D
    X_train = X_train.reshape(-1, 28, 28, 1)
    X_test = X_test.reshape(-1, 28, 28, 1)
    X_train = X_train/255.0
    X_test = X_test/255.0
    N_TRAIN = X_train.shape[0]
    N_TEST = X_test.shape[0]
    DIMS = [X_train.shape[1], X_train.shape[2]]
    # Shuffle data
    order_train = random.permutation(N_TRAIN)
    order_test = random.permutation(N_TEST)
    X_TRAIN = X_train[order_train,:] # Shape 60000x28x28
    Y_TRAIN = labels_train[order_train] # Shape 60000x1
    X_TEST = X_test[order_test,:] # Shape Nx28x28
    Y_TEST = labels_test[order_test] # Shape Nx1

    #endregion

    # Prepare grid search over all combinations of hyperparameters
```

```
    num_param_states = len(FILTERS) * len(LAYER_DEPTHS) * len(HIDDEN_LAYERS) *
        len(BATCH_SIZES) * len(OPTIMIZERS)
    combinations = list(itertools.product(FILTERS, LAYER_DEPTHS, HIDDEN_LAYERS, BATCH_SIZES,
        OPTIMIZERS))
    # Append indices to this large list
    combinations = [(x + (i,)) for i, x in enumerate(combinations)]

    # Prepare gigantic list of complete parameters.
    args_full = [(combo, DIMS, ACTIVATION_FUNCTIONS, LOSS_FUNCTION, X_TRAIN, Y_TRAIN, EPOCHS,
        kf, n_folds) for combo in combinations]

    print(f"ready to iterate over {num_param_states} combinations")

    # Start running parallel trainings over all combinations
    outputData = []
    with multiprocessing.Pool() as pool:
        results = pool.starmap(eval_model, args_full)

        # Open the CSV file once and write all results
        with open(filename, 'w', newline='') as csvfile:
            writer = csv.writer(csvfile, delimiter=',')
            header = [["Index"], ["Filters"], ["Depth"], ["Layers"], ["Batch Size"],
                ["Optimizer"], ["Valid Loss"], ["Valid Acc"], ["Time"]]
            writer.writerow(header)
            for result in results:
                outputData.append(result)
                writer.writerow(result)

    # Pick top 1 in terms of validation loss and get test error
    sorted_outputData = sorted(outputData, key=lambda x: x[5])
    top_1 = sorted_outputData[0]
    filters, index, depth, hLayers, batchSize, optimizer, _, _, _ = top_1
    result = test_model((filters, depth, hLayers, batchSize, optimizer, index), DIMS,
        ACTIVATION_FUNCTIONS, LOSS_FUNCTION, X_train, y_train, X_test, y_test, EPOCHS)
    print("Filters, Index, Depth, Layers, Batch Size, Optimizer, Test Loss, Test Accuracy,
        Training Time")
    print(result)

# Function to train and evaluate TF model
def eval_model(dynamic_args, d, activation, loss, X_train, y_train, epochs, kf, n_folds):
    # Unpack arguments
    filters, depth, hLayers, batchSize, optimizer, index = dynamic_args
    # Clear previous models and graphs
    tf.keras.backend.clear_session()
    # Create a new graph for the current process
    with tf.Graph().as_default():
        # Create a session for the current graph
        with tf.compat.v1.Session() as sess:
            # Remind Tensorflow to keep quiet
            tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)

            # Model creation
            if hLayers == 1:
                model = tf.keras.models.Sequential([
                    # Convolutional layer with N filters, kernel size of 3x3, ReLU activation,
                        and input shape for MNIST
                    tf.keras.layers.Conv2D(filters, kernel_size=(3, 3), activation=activation,
                        input_shape=(d[0], d[1],1)),
                    # Pooling layer to reduce dimensionality
                    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
                    # Flatten the output to feed into the dense layer
                    tf.keras.layers.Flatten(),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(10, activation='softmax')
                ])
            else:
                model = tf.keras.models.Sequential([
```

```
                    # Convolutional layer with N filters, kernel size of 3x3, ReLU activation, and input shape for MNIST
                    tf.keras.layers.Conv2D(filters, kernel_size=(3, 3), activation=activation, input_shape=(d[0], d[1],1)),
                    # Pooling layer to reduce dimensionality
                    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
                    # Flatten the output to feed into the dense layer
                    tf.keras.layers.Flatten(),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(10, activation='softmax')
                ])

            # Compile the model
            model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

            # Start stopwatch
            start_time = time.time()

            # Initialize averaged results, test results are to be kept aside.
            valid_accuracy = 0
            valid_loss = 0
            # Train the model
            for index_train, index_val in kf.split(X_train):
                X_vtrain, X_val = X_train[index_train], X_train[index_val]
                y_vtrain, y_val = y_train[index_train], y_train[index_val]
                history = model.fit(X_vtrain, y_vtrain, epochs=epochs, batch_size=batchSize, validation_split=0.2)
                # Test this model on the current fold's validation dataset, and the test set
                this_valid_loss, this_valid_accuracy = model.evaluate(X_val, y_val)

                # Add to average
                valid_accuracy += this_valid_accuracy / n_folds
                valid_loss += this_valid_loss / n_folds

            # Stop stopwatch
            end_time = time.time()


            training_time = end_time - start_time
            print(f"Done with Run {index}, t = {training_time}")
            return index, filters, depth, hLayers, batchSize, optimizer, valid_loss, valid_accuracy, training_time

if __name__ == "__main__":
    main()
```

```
## Code Execution Steps:
# - If you haven't already, run "pip3 install numpy matplotlib seaborn torch torchvision multiprocessing tensorflow scikit-learn ucimlrepo itertools pandas csv"
# - Modify individual hyperparameters to evaluate them on the test set.
# - Run python file using any Python >3.9 interpreter with the above libraries installed.
import numpy as np
import numpy.random as random
import matplotlib.pyplot as plt
import seaborn as sns
import torch, torchvision
import os
import tensorflow as tf
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'  # Suppresses noisy outputs
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)
from sklearn.metrics import confusion_matrix, classification_report
import pandas as pd
import csv
import time

```

```
def main():
    ## Hyperparameters
    filters = 64
    layer_depth = 80
    hidden_layers = 1
    epochs = 30
    batch_size = 64
    activation_function = "relu"
    loss_function = "sparse_categorical_crossentropy"
    optimizer = "adam"

    filename = "CNN Final Results.csv"

    ## Data Pre-Processing
    #region
    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()
    # Add a fourth dimension to satisfy conv2D
    X_train = X_train.reshape(-1, 28, 28, 1)
    X_test = X_test.reshape(-1, 28, 28, 1)
    X_train = X_train/255.0
    X_test = X_test/255.0
    N_TRAIN = X_train.shape[0]
    N_TEST = X_test.shape[0]
    DIMS = [X_train.shape[1], X_train.shape[2]]
    # Shuffle data
    order_train = random.permutation(N_TRAIN)
    order_test = random.permutation(N_TEST)
    X_train = X_train[order_train,:] # Shape 60000x28x28
    y_train = labels_train[order_train] # Shape 60000x1
    X_test = X_test[order_test,:] # Shape Nx28x28
    y_test = labels_test[order_test] # Shape Nx1

    #endregion

    result = test_model((filters, layer_depth, hidden_layers, batch_size, optimizer), DIMS,
        activation_function, loss_function, X_train, y_train, X_test, y_test, epochs)
    filters, depth, hLayers, batchSize, optimizer, test_loss, test_accuracy, training_time,
        conmat, y_pred = result

    print("Filters, Depth, Layers, Batch Size, Optimizer, Test Loss, Test Accuracy, Training
        Time")
    print(f"{filters}, {depth}, {hLayers}, {batchSize}, {optimizer}, {test_loss},
        {test_accuracy}, {training_time}")

    # Print confusion matrix
    # print("Confusion Matrix:")
    # print(conmat)


    plt.figure(figsize=(10,8))
    sns.heatmap(conmat, annot=True, fmt='d', cmap='Blues')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title('Confusion Matrix for CNN Classifier')
    plt.show()

    # Compute and print classification report
    # target_names = ['Class 0', 'Class 1']
    # report = classification_report(y_test, (y_pred > 0.5), target_names=target_names)
    # print("Classification Report:")
    # print(report)

```

```python
# Function to train and evaluate TF model
def test_model(dynamic_args, d, activation, loss, X_train, y_train, X_test, y_test, epochs):
    # Unpack arguments
    filters, depth, hLayers, batchSize, optimizer = dynamic_args
    # Clear previous models and graphs
    tf.keras.backend.clear_session()
    # Create a new graph for the current process
    with tf.Graph().as_default():
        # Create a session for the current graph
        with tf.compat.v1.Session() as sess:
            # Remind Tensorflow to keep quiet
            tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)

            # Model creation
            if hLayers == 1:
                model = tf.keras.models.Sequential([
                    # Convolutional layer with N filters, kernel size of 3x3, ReLU activation, and input shape for MNIST
                    tf.keras.layers.Conv2D(filters, kernel_size=(3, 3), activation=activation, input_shape=(d[0], d[1],1)),
                    # Pooling layer to reduce dimensionality
                    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
                    # Flatten the output to feed into the dense layer
                    tf.keras.layers.Flatten(),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(10, activation='softmax')
                ])
            else:
                model = tf.keras.models.Sequential([
                    # Convolutional layer with N filters, kernel size of 3x3, ReLU activation, and input shape for MNIST
                    tf.keras.layers.Conv2D(filters, kernel_size=(3, 3), activation=activation, input_shape=(d[0], d[1],1)),
                    # Pooling layer to reduce dimensionality
                    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
                    # Flatten the output to feed into the dense layer
                    tf.keras.layers.Flatten(),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(depth, activation=activation),
                    tf.keras.layers.Dense(10, activation='softmax')
                ])

            # Compile the model
            model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

            # Start stopwatch
            start_time = time.time()

            # Train the model
            history = model.fit(X_train, y_train, epochs=epochs, batch_size=batchSize, validation_split=0.2)
            test_loss, test_accuracy = model.evaluate(X_test, y_test)

            # Predict the test set
            y_pred = model.predict(X_test)
            y_pred_labels = np.argmax(y_pred, axis=1)

            # Compute confusion matrix
            conmat = confusion_matrix(y_test, y_pred_labels)

            # Stop stopwatch
            end_time = time.time()

            training_time = end_time - start_time
            print(f"Done with test, t = {training_time}")
            return filters, depth, hLayers, batchSize, optimizer, test_loss, test_accuracy, training_time, conmat, y_pred
```

```
143
144 if __name__ == "__main__":
145     main()
```

REFERENCES

[1] E. Alpaydin, “Introduction to Machine Learning,” The MIT Press. Cambridge, Second Edition, pp. 1–20, 2010.

[2] W. Wolberg, O. Mangasarian, N. Street, W. Street, “Breast Cancer Wisconsin (Diagnostic),” UC Irvine Machine Learning Repository, Available: https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic [Accessed: Dec. 02, 2023]

[3] B. Becker, R. Kohavi, “Adult,” UC Irvine Machine Learning Repository, Available: https://archive.ics.uci.edu/dataset/2/adult [Accessed: Dec. 02, 2023]

[4] H. Xiao, K. Rasul, R. Vollgraf, “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms,” CoRR, Vol. 17080.07747. Available: http://arxiv.org/abs/1708.07747 [Accessed: Dec. 02, 2023]

[5] S. Abdulhamit. “Machine Learning Techniques,” Practical Machine Learning for Data Analysis Using Python, Academic Press, 2020, pp. 91-202.

[6] F. Pedregosa et al., ”Scikit-learn: Machine Learning in Python,” JMLR 12, pp. 2825-2830, 2011.

[7] C.R. Harris, K.J. Millman, S.J. van der Walt, et al. ”Array programming with NumPy,” Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2. Available: https://www.nature.com/articles/s41586-020-2649-2 [Accessed: Dec. 05, 2023]

[8] ”itertools — Functions creating iterators for efficient looping,” Pyhton 3.12.0 Documentation, Available: https://docs.python.org/3/library/itertools.html [Accessed: Dec. 05, 2023]

[9] The Pandas Development Team, ”pandas-dev/pandas: Pandas,” Zenodo, feb 2020, 10.5281/zenodo.3509134, Available: https://doi.org/10.5281/zenodo.3509134 [Accessed: Dec. 05, 2023]

[10] J. D. Hunter, ”Matplotlib: A 2D Graphics Environment”, Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.

[11] M. L. Waskom, ”seaborn: statistical data visualization,” Journal of Open Source Software, 2021, 6(60), 3021, https://doi.org/10.21105/joss.03021 [Accessed: Dec. 05, 2023]

[12] OpenAI, ”ChatGPT: Language Model for Natural Language Processing,” [Online], Available: https://www.openai.com/ [Accessed: Dec. 05, 2023]

[13] M. Abadi, A. Agarwal, P. Barham, et al., ”TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems,” 2015, Available: https://www.tensorflow.org/ [Accessed: Dec. 05, 2023]

[14] S. Marcel, Y. Rodriguez, ”Torchvision the Machine-Vision Package of Torch,” in Proceedings of the 18th ACM International Conference on Multimedia, Association for Computing Machinery, Firenze, Italy, 2010, pp. 1485-1488. Available: https://doi.org/10.1145/1873951.1874254 [Accessed: Dec. 05, 2023]