Berfin Kavşut - 21602459
Mert Ertuğrul - 21703957

EEE 485/585 SPRING 2021
TERM PROJECT : Phase II Report

1. Problem Statement

This project aims to demonstrate the development and performance of a number of machine learning algorithms on the task of binary classification for heart disease based on a feature set consisting of biomarkers. The core machine learning algorithms are;

- LASSO Classification
- Neural Networks
- K Nearest Neighbours
- Support Vector Machines

First and foremost, we work with four core algorithms that are commonly used for binary classification problem. K-Nearest Neighbours algorithm is a straightforward algorithm and does not need training a model. It may not be compatible with higher dimensional feature spaces, yet its application after feature selection and extraction can yield improved results [1]. SVM is again a popular algorithm for binary classification [2]. Neural Networks are known as high performers for highly complex data with non-linear decision boundaries and can approximate to any continuous decision boundary. Lastly, LASSO classification is efficient for fitting simpler relationships while also producing interpretable feature weights.

Through implementing and comparing our models, our aim is to understand the nature of our problem and the behavior of our ML models under the constraints of a small but high dimensional dataset. Among our models, Neural Networks have the highest number of parameters, which means that they require larger datasets for sufficient performance. Given the size of our dataset, we expect SVM, which does not involve as many parameters as neural networks, and KNN, the non-parametric model, to outperform neural networks. However, if the underlying decision boundary of our problem is highly complex, the appropriate model choice could be neural networks, albeit the dataset size would be a limiting factor. If, on the other hand, certain features are redundant and hindering performance, LASSO may reveal this.

We also apply PCA to our data and form a second dataset combining the original features with principal components and then compare the performance of the two datasets for every model.

2. Dataset Description

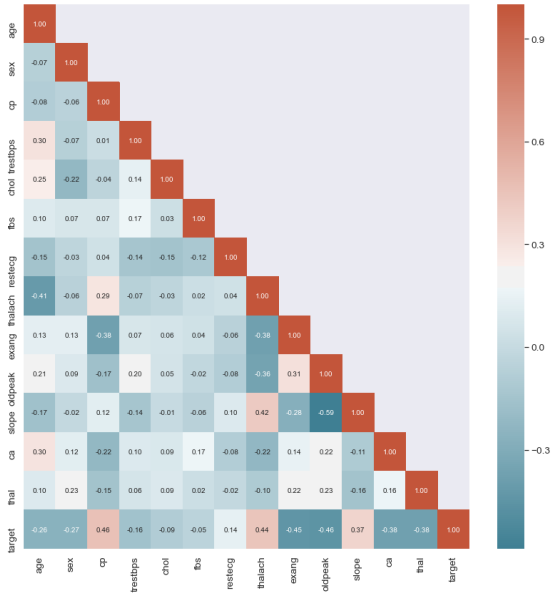
The dataset used for this project is “Heart Disease Data Set” from “UCI Machine Learning Repository” The dataset consists of 303 data instances. Although the original dataset provides 75 features, the widely used version provides 13 features per instance that are recommended by the repository [3]. We have used all 13 features for the training and evaluation of all of our algorithms, later evaluating the importance of each feature. A list of the 13 features are given below:

1. Age: age in years
2. Sex: (1 = male; 0 = female)
3. Chest pain type: (typical angina, atypical angina, non-anginal pain, asymptomatic)
4. Resting blood pressure (mm Hg)
5. Serum cholesterol (mg/dl)
6. Fasting blood sugar ((> 120 mg/dl) (1 = true; 0 = false))
7. Resting electrocardiographic results (normal, having ST-T wave abnormality, left ventricular hypertrophy)

8. Maximum heart rate achieved
9. Exercise induced angina (1 = yes; 0 = no)
10. ST depression induced by exercise relative to rest
11. The slope of the peak exercise ST segment (usnloping, flat, downsloping)
12. Number of major vessels (0-3) colored by fluoroscopy
13. Thallium : (3 = normal; 6 = fixed defect; 7 = reversible defect)

Target: Has heart disease or not (1=yes, 0=no)

Table 1: Lower Triangular Covariance Matrix of the Dataset



To explore the properties of the dataset such as the existence of missing values and the spread of various combinations of features, we used the Pandas library of Python and Pandas Profiling Report specifically. The features that have the highest absolute correlation with the target are:

- cp (Chest pain type)
- exang (Exercise induced angina)
- oldpeak (ST depression induced by exercise relative to rest)
- thalanch (Maximum heart rate achieved)
- ca (Number of major vessels (0-3) colored by fluoroscopy)
- thal (Thallium)

As can be seen from the covariance matrix, except for the pair of "oldpeak" and "slope", most features are not highly correlated and there are no obvious feature pairs that behave similarly against any other feature.

We will later see how these correlation values measure up in terms of importance when LASSO is applied. Some will remain important while others will be given small weights despite having high correlation.

3. Preprocessing

The dataset is shuffled and standardized (converted to zero mean and unit variance) before being used for each algorithm. For standardization, numerical features and categorical features that are not binary and have a meaningful order between categories were standardized. Binary category features were not standardized.

Table 2: Statistical Properties of the Dataset

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000
mean	54.366337	0.683168	0.966997	131.623762	246.264026	0.148515	0.528053	149.646865	0.326733	1.039604	1.399340	0.729373	2.313531	0.544554
std	9.082101	0.466011	1.032052	17.538143	51.830751	0.356198	0.525860	22.905161	0.469794	1.161075	0.616226	1.022606	0.612277	0.498835
min	29.000000	0.000000	0.000000	94.000000	126.000000	0.000000	0.000000	71.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	47.500000	0.000000	0.000000	120.000000	211.000000	0.000000	0.000000	133.500000	0.000000	0.000000	1.000000	0.000000	2.000000	0.000000
50%	55.000000	1.000000	1.000000	130.000000	240.000000	0.000000	1.000000	153.000000	0.000000	0.800000	1.000000	0.000000	2.000000	1.000000
75%	61.000000	1.000000	2.000000	140.000000	274.500000	0.000000	1.000000	166.000000	1.000000	1.600000	2.000000	1.000000	3.000000	1.000000
max	77.000000	1.000000	3.000000	200.000000	564.000000	1.000000	2.000000	202.000000	1.000000	6.200000	2.000000	4.000000	3.000000	1.000000

When we inspect the statistical properties of different features, we see that the means and standard deviations of numerical features vary drastically and categorical features exist as well. Therefore we standardize the relevant features. The data samples were already given as sufficiently balanced:

Instances with heart disease: 165 (54.46%)
Instances without heart disease: 138 (45.54%)

Therefore, the dataset was not further preprocessed to account for any skewed sample number issues.

3.1. PCA [2]

PCA is an unsupervised learning algorithm. It is used mainly for two reasons: dimensionality reduction and data visualization. Dimensionality reduction is a remedy for the curse of dimensionality. Curse of dimensionality means machine learning models cannot learn on high-dimensional data because data points start to get closer with respect to euclidean distance. Therefore, a representation on low-dimensional space is needed for high-dimensional data while preserving the maximum variance. Usually, PCA is used as a preprocessing step before supervised learning algorithms. In our case, we used PCA as a feature engineering tool to see if we can create more optimal features than the original ones.

$$\text{Total variance: } \Sigma = \frac{1}{2} X^T X, where\ X\ is\ nxp\ dataset$$

X is the standardized dataset. The aim is to decrease the dimension from p to k. Principal component loading vectors are the eigenvectors of $\Sigma$and we choose k eigenvectors with the largest eigenvalues. Loading vectors, which are orthogonal and have unit norm, are used as projection vectors. Variance is maximized in the direction of the loading vectors. Principal component scores (z) are found by the inner product of data points and the loading vectors, i.e. projection operation.

For choosing the k value, the proportion of variance explained (PVE) needs to be calculated. It is desired that at least 80% of total variance is captured after PCA. As can be seen from our cumulative variance graph (PVE(first k)), 80% is obtained at about 8 principal components, therefore we only used the first 8 components throughout the project.

$$PVE(m) = \frac{\frac{1}{n}\sum_{i=1}^{n}\left(\sum_{j=1}^{p} x_{ij} u_{mj}\right)^2}{\frac{1}{n}\sum_{j=1}^{p}\sum_{i=1}^{n} x_{ij}^2}$$

The term on the numerator is the variance explained by the mth component, the term on the denominator is the total variance. PVE for the first k principal components can be found by summing up PVE(m) for m=1,...,k. Explained variance plots are in Figure 1 (left side).

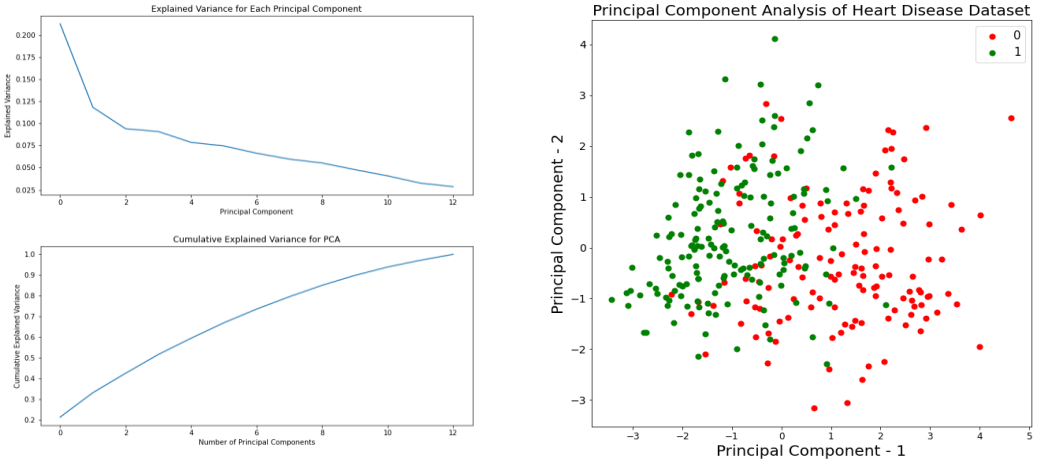


Figure 1: (Left side) Explained Variance Graphs, (Right side) PCA Plot

As mentioned before, PCA can be used for data visualization. In Figure 1 (right side), each data point is projected on 2-dimensional space with the base vectors of the first 2 loading vectors. As seen in Figure 1 (right side), there is a visual separation between the data points with different labels; however, data from the two labels is mixed at the boundary. Many points fall into the area of the other class, which means higher dimensions are needed to have a clear decision boundary with linear classifiers.

4. Models & Implementation

4.1. LASSO Classification

4.1.1. Model Description

LASSO (least absolute shrinkage and selection operator) regression is a linear regression method with regularization. It is used for two main tasks: regularization (shrinkage) and feature selection, where it gets its name from [4]. When a threshold is set to binarize the output value, LASSO can be used for classification as well, which is how we use it for this project.

$$\min_{\beta} \sum_{i=1}^{n} \left(y_i - \boldsymbol{x_i^T \beta}\right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| = RSS(\beta) + \lambda \sum_{j=1}^{p} |\beta_j|$$

The aim is to minimize loss function with the penalty term of $l1$-norm of the coefficient vector, which means the LASSO problem is an $l1$ optimization problem. In the minimization problem, λ is the tuning parameter determining the strength of regularization. By regularization, the coefficient vector β is shrunk [2]. Generally, shrinkage is the reason why LASSO gives better accuracy results due to low variance with very small increase in bias. Hence, it is preferred for data with small number of samples and high number of features [4].

When $\lambda = 0$, the LASSO solution is the ordinary least squares solution. Model complexity is high when there is no regularization and variance is high. As λ goes to zero, model complexity decreases and bias increases. There is a trade-off between bias and variance for the tuning parameter selection. The tuning parameter is usually selected by cross validation [2].

Due to the hypercubic geometry of $l1$-norm solutions, β becomes a sparse vector with zeros as λ increases. Actually, this is the reason why LASSO is used as a feature selector. LASSO aids to model interpretability by showing which predictors are important or not. Feature selection can be used to avoid overfitting [4]. Matrix derivation is not possible for LASSO solution, therefore there is no closed form solution like ordinary least squares solution. The solution can be found by gradient descent [2].

4.1.2. Implementation Details

- Stochastic gradient descent is implemented to learn the weight updates.
- When a weight update is calculated, the sign of the weight is checked. If the weight is positive, the weight update becomes the derivative of residual sum of squares plus regularization parameter. When the weight is negative, the weight update becomes the derivative of residual sum of squares minus regularization parameter.

4.2. Neural Network

4.2.1. Model Description

In our project we used Multilayer Perceptrons (MLP), which is a type of a feedforward neural network. MLP consists of neuron units. The weighted summation of inputs are calculated in a neuron unit, and the

neuron is activated based on the output and activation function. If the output value is below a certain threshold, the neuron is not activated, and vice versa [5].

MLPs consist of input, hidden and output layers. Input layer feeds the input to the hidden layers. Hidden layers have neurons with non-linear activation functions and each output of a hidden layer is fed to the next hidden layer. Finally, the last hidden layer is fed to the output layer. MLPs are used for classification, recognition, prediction and approximation. The power of MLP comes from its nonlinear activation functions. While perceptrons can be used only for linearly separable problems, MLPs can be used for nonlinear separation problems, which means they can approximate any continuous decision boundary [6].

MLPs are trained by gradient descent. It aims to find the global minimum of the loss function based on the error between real and desired outputs. Gradient descent guarantees to find a local minimum but not the global minimum. In forward propagation, layer outputs are calculated. In backward propagation, gradients are calculated and weights are updated [6].

4.2.2. Implementation Details

- "NeuralNetwork" class stores and operates on the layers during forward and backward propagation. It takes learning rate, number of epochs, classification threshold, and mini-batch size as parameters. Number of layers is also a parameter of the model to determine.
- For loss computation, we used cross entropy loss. For activation functions, we used sigmoid activation function after the output layer as it is commonly used for producing an output value between 0 and 1 for binary classification. ReLU was used after the hidden layers.
- Mini batch size parameter is set to zero for batch, one for stochastic and other numbers for mini-batch gradient descent.

4.3. K-Nearest Neighbours

4.3.1. Model Description

K-Nearest Neighbours (KNN) is a supervised machine learning algorithm based on using the distances between data points in the feature space according to a specified distance metric. The algorithm determines k neighbouring data points that have the smallest distance to the data point we aim to label. Afterwards, it classifies the new data point based on the majority vote of the neighbouring points' labels. The algorithm is simple compared to the other algorithms we worked with since it is non-parametric and makes predictions without a separate stage of training. All available data points are used at one shot to classify new data, which means that as the dataset becomes larger, the algorithm becomes much slower. Given the small size of our dataset (303 samples) we did not consider this to be a major setback for us [1].

4.3.2. Implementation Details

- "KNN_classify" is implemented as a function since there is no parameter to learn for the model, therefore it was not implemented as a class.
- "KNN_classify" takes a set of reference data corresponding to training data and testing data as well as an option to choose the value k and the type of distance metric.
- Euclidian (L2), manhattan (L1) and cosine distance metrics were included as options.

4.4. Support Vector Machine

4.4.1. Model Description

Support vector machines serve a similar purpose to perceptrons as linear classifiers. Perceptrons do not guarantee that the separating hyperplane will be the best hyperplane in terms of maximizing the distance to each data instance in total. Perceptrons can yield infinitely many possible linear separators whereas the separating hyperplane of SVM is unique [7].

SVM aims to find the hyperplane which has the highest distance to each of the data points while keeping them on the correct side for classification. The linear separator is calculated by support vectors. SVM depends on a small subset of data points which are closest to the separating hyperplane. These data points are called support vectors since the hyperplanes can be thought of as being supported by these support vectors. [8].

SVM is an extension of support vector classifier, which is an extension of maximal margin classifier. Margin means the distance between the hyperplane and the closest data point on each side i.e. class. Maximal margin classifiers directly find the hyperplane with the maximal margin and do not have any tolerance against noisy data points. Therefore, they may lead to low bias and high variance, i.e. overfitting. Support vector classifiers allow some of the data points to be on the wrong side of the hyperplane instead of seeking the largest possible margin. SVMs use linear classifiers to produce non-linear decision boundaries so that our dataset does not need to be linearly separable. For this purpose, feature space can be enlarged by using kernels, e.g. polynomial kernels [1].

$$\min_{w,b} \|\boldsymbol{w}\|^2 + C \sum_{i=1}^{n} \max\left(1 - y_i(\boldsymbol{w}^T \boldsymbol{x_i} + b), 0\right)$$

$\|\boldsymbol{w}\|^2$is l_2 regularizer and $\max\left(1 - y_i(\boldsymbol{w}^T \boldsymbol{x_i} + b), 0\right)$ term is the hinge loss. y_i is -1 for negative class and 1 for positive class. C is the bugdet determining how much to sacrifice to have a simpler and general solution. The cost function which is hinge loss function with regularization term is minimized by Gradient Descent. When C is small, the model may lead to overfitting like a maximal margin classifier. When C increases, we are more tolerant to the violation of margins and we may have higher bias but lower variance, which means our model can generalize to the test set. To sum up, C value determines the trade-off between bias and variance [2].

4.4.2. Implementation Details

- "SVM" stores the regularization parameter C and weight vector w.
- Model is trained by stochastic gradient descent to minimize the cost function. Number of epochs and learning rate are parameters to determine.

5. Results and Analysis

5.1. Parameter Search, Validation and Testing Methods

Firstly, the dataset is divided into 90% training and 10% test sets. All procedures that may leak information to the test set such as standardization, PCA and parameter search are based solely on the training set. For example, the standardization mean and standard deviation of the training set were used for the test set. For each model, optimal model parameters were determined by carrying out grid search over common parameter values by applying k-fold cross validation on the training set with 5 for k. When doing grid search, the best parameter set was chosen according to average cross validation accuracy. Relevant graphs and values resulting from the cross validation were stored. Further parameter search details will be explained later in the report. Using the best parameters for each model, the models were tested on the test set.

We evaluate the test set classification performance more thoroughly by also including precision, recall, and F1-score values. Metrics are defined below:

- Accuracy = (TP+TN)/(TP+TN+FP+FN)
- Precision = TP/(TP+FP)
- Recall = TP/(TP+FN)
- F1-Score = TP/(TP + 0.5*(FP+FN))

5.2. LASSO Classification

LASSO was used both as a classifier and to observe the importances of the features. First we explain the classifier results.

5.2.1. Results for Classification

Number of training epochs, learning rate and lambda were determined through grid search with cross validation on training data. For the original dataset without PCA, the optimal set of parameters were found as given on the left below. The best cross validation accuracy using the best parameters on the training set was **0.909**. The test performance was obtained by training the model on the entire training set with the best parameters and applying the test set to it. The test metrics are given on the right below.

Num. of epochs = 3	Lambda = 0	Accuracy = 0.80	Recall = 0.79
Learning rate = 0.01		Precision = 0.86	F1-Score = 0.78

The optimal lambda was zero, meaning that regularization did not improve accuracy for the original dataset and linear regression was the final model. This may indicate that eliminating features is not likely to improve performance for our models. Once the 8 principal components were added to the features, cross validation accuracy on the training set increased to **0.927** and the optimal lambda became nonzero as seen on the left below. Despite these changes, test set performance metrics did not change at all (on the right side below).

Num. of epochs = 5	Lambda = 0.01	Accuracy = 0.80	Recall = 0.79
Learning rate = 0.001		Precision = 0.86	F1-Score = 0.78

Test results are identical despite parameter differences and different weights for features (some principal components were given significant weights). The model may be reaching the same local minimum with different parameters and feature combinations. Also, the overall dataset size and test set size (about 30) are small enough that numbers such as test results can be identical. There may be a specific group of outlier data points that are misclassified by most models no matter how we optimize the parameters.

In Figure 2 below, from the first weight change graph, we see that “restecg” increased in importance despite not having high correlation with the label compared to many other features such as “cp” and “oldpeak”, both of which have high weights matching their correlations with the label. “Chol”, “fbs” and “age” seem to be almost eliminated but zeroed out. Overall, resting blood pressure (restecg), number of major vessels coloured by fluoroscopy (ca) and chest pain type (cp) appear to be important criteria for classifying heart disease status. When the principal components are added, the whole picture changes. The first few principal components are given high weights, some of the previously important features remain important but not as much as they were before PCA and others such as “thalach” are brought below 0.05 despite being above 0.15 before PCA. Newly added principal components likely explain away much of the information previously only provided by the original features and so some of those features are no longer important. This was an expected outcome. Other features such as “fbs” have risen in importance from an absolute weight of 0.007 to 0.166 so there may be useful information in such features that was not expressed in the model previously.

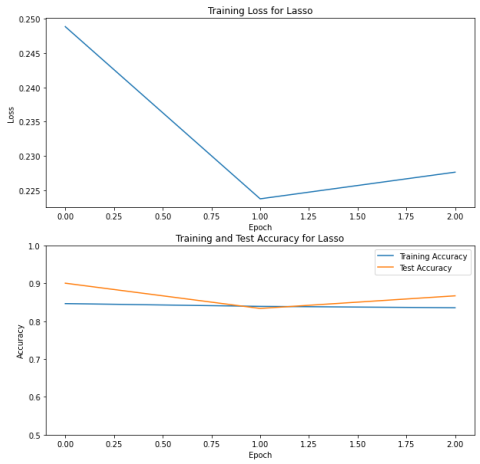
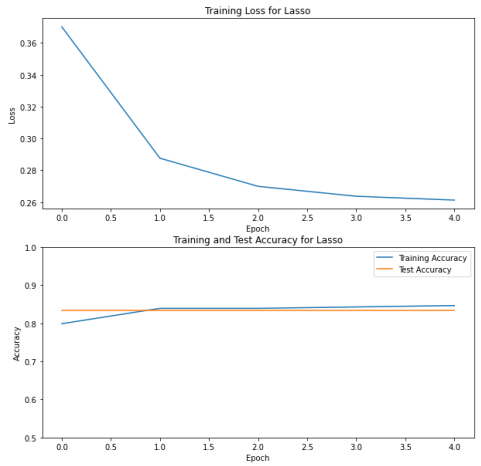
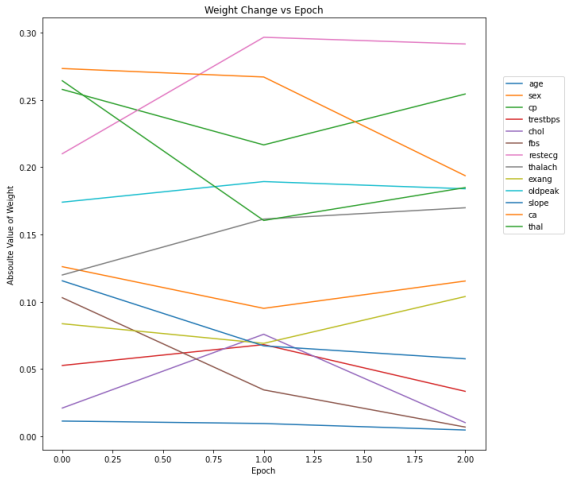
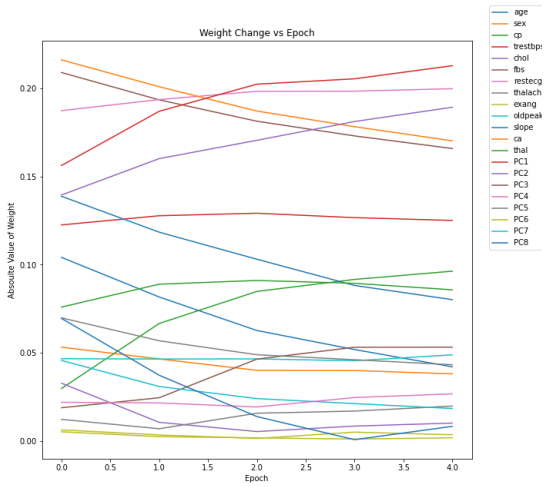
	Without Adding Principal Components to the Dataset	**With 8 Principal Components Added to the Dataset**
Loss and Accuracy Graphs		
Feature Weights vs Epochs for Each Feature		

Figure 2: Result Graphs for LASSO Regression

5.2.2. Results for Varying Lambda

Among the original features and principal components, all but one (PC1) are zeroed out after lambda reaches 0.3. “ca”,”PC2” and “restecg” follow “PC1” in importance. As we saw previously, almost all features remained non-zero for the optimal lambda of 0.01 and this graph shows most features zeroing out at around the same lambda so eliminating features does not appear to be an effective way of increasing performance.

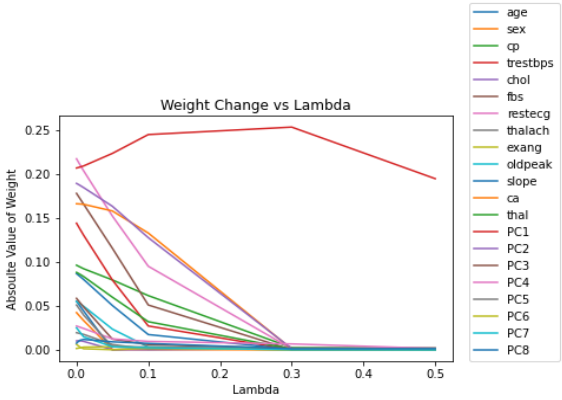


Figure 3: Weight Change vs. Lambda Plot

5.3. Neural Networks

For the original dataset without PCA, the optimal set of parameters was found as given on the left below. The best cross validation accuracy using these parameters on the training set was **0.846**. The test performance was obtained by training the model on the entire training set with the best parameters and applying the test set to it. The test metrics are given on the right below.

Num. of epochs = 25	Num. of hidden layers = 1	Accuracy = 0.9	Recall = 0.91
Learning rate = 0.5	Mini-batch size = 0 (batch gradient descent)	Precision = 0.91	F1-Score = 0.9

We must note that the cross-validation training set accuracy did not vary significantly between indifferent parameter combinations. The learning rate found is much higher than conventional values used for neural networks such as 0.001, 0.01, 0.1. This parameter is inversely correlated with mini batch size and related to sizes used in weight initialization which may be the reason behind the high learning rate. Still, the test accuracy of 0.9 is the best we have obtained for any model in the project. Afterwards, we added the 8 most explanatory principal components to both the training and test sets. The cross-validation training set accuracy increased to **0.85**, which was not reflected in the test set results. Here are the relevant optimal parameters found and the test metrics:

Num. of epochs = 10	Num. of hidden layers = 1	Accuracy = 0.87	Recall = 0.87
Learning rate = 0.1	Mini-batch size = 20	Precision = 0.87	F1-Score = 0.87

It appears that adding principal components to the existing set of features did not improve test performance. The small difference combined with the change in the parameter set means that we cannot definitively say that the additional features hinder performance in general, but may have altered the parameter selection process in a random way to result in these values.

Loss and Accuracy Graphs Without Adding Principal Components to the Dataset	**Loss and Accuracy Graphs With 8 Principal Components Added to the Dataset**

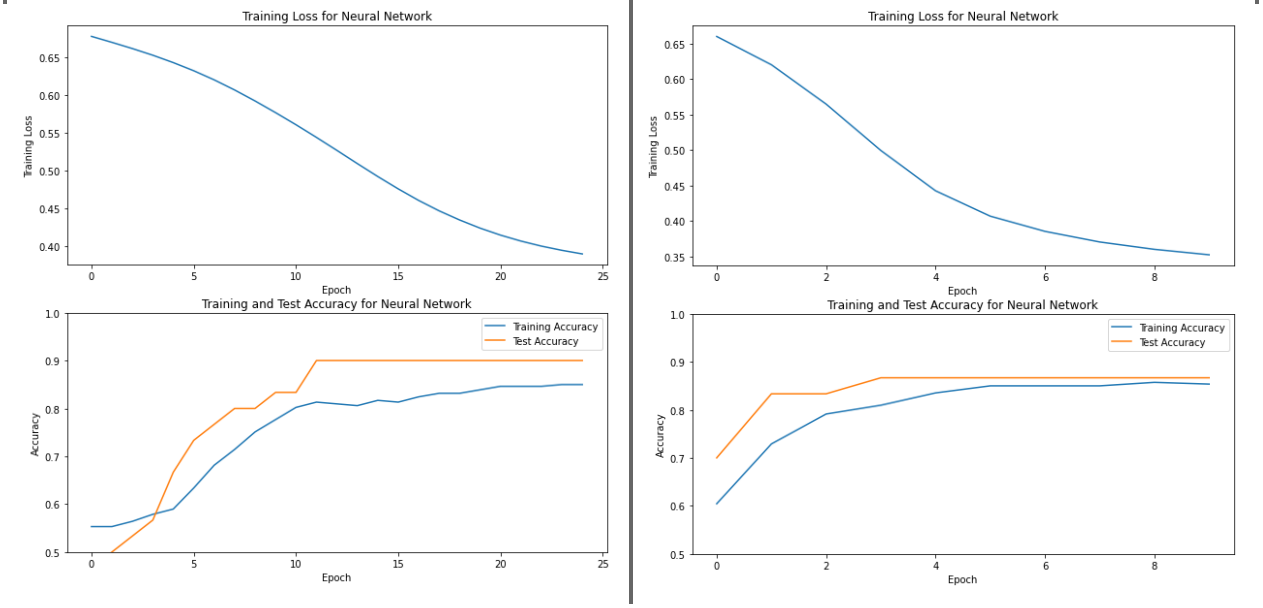


Figure 4: Result Graphs for Neural Network

5.4. KNN

For the original dataset without PCA, the optimal parameters were found as given on the left below. The best cross validation accuracy using these parameters on the training set was **0.867**. The test performance metrics are given on the right below.

k = 13 Distance = manhattan	Accuracy = 0.87 Precision = 0.87 Recall = 0.87 F1-Score = 0.87

After adding the principal components, the cross-validation training set accuracy did not change, neither did the test result metrics. However, the ideal number of neighbours changed to 11 and the distance metric became cosine. The fact that the results were the same with a different dataset and parameters is quite unusual, perhaps indicating that most models find the same inherent patterns that produce the same accuracy results. When we look at the graphs, it appears that adding the PCA drops the accuracy for manhattan distance metric to levels similar to euclidean distance while accuracy with cosine distance is completely unaffected, which explains the change in optimal distance metric.

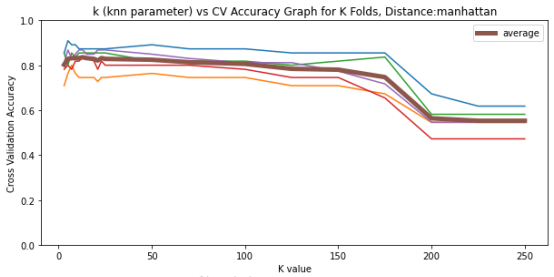
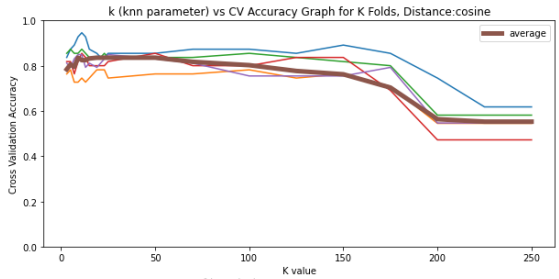
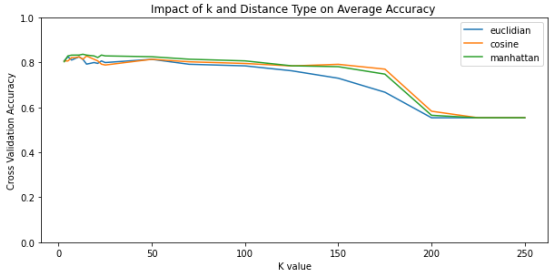
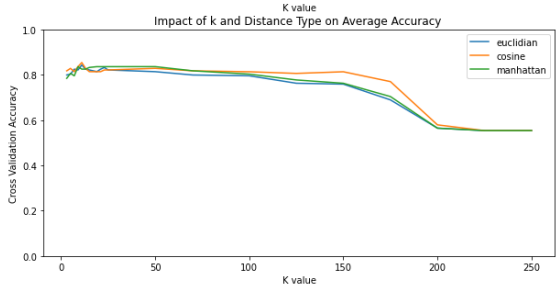
	Without Adding Principal Components to the Dataset	**With 8 Principal Components Added to the Dataset**
Accuracy vs K for each fold of 5-fold cross validation (optimal distance metric)		
Average Accuracy of 5 folds vs K value for each distance metric		

Figure 5: Cross Validation Parameter Search Graphs for K Nearest Neighbours

5.5. SVM

For the original dataset, the optimal parameters are given on the left below. The best cross validation accuracy using these parameters on the training set was **0.831** which was not reflected in the test set results. The test set performance metrics are given on the right below.

Num. of epochs = 50 Learning rate = 0.0001	Regularization param. = 5	Accuracy = 0.77 Precision = 0.78	Recall = 0.76 F1-Score = 0.76

After adding the principal components, the cross validation training set accuracy increased to 0.846. More importantly, the test set performance metric improved all across the board as given on the right below. However, although not displayed here, precision for the negative class dropped significantly (from 0.82 to 0.64) while recall for the negative class increased (from 0.64 to 0.83). This is caused by a significant increase in false positives and a drop in false negatives. Here are the relevant optimal parameters found

and the test metrics:

Num. of epochs = 10	Regularization param. = 5	Accuracy = 0.87	Recall = 0.85
Learning rate = 0.0001		Precision = 0.79	F1-Score = 0.81

Looking at the test metrics above and the graphs below, we can see that before introducing the PCA features, training performance was not reflected in the test performance indicating that the model could not generalize to the test set very well. After introducing principal components, the model no longer overfits and test performance is on par with training but the model may be too biased towards the positive class.

Loss and Accuracy Graphs Without Adding Principal Components to the Dataset	**Loss and Accuracy Graphs With 8 Principal Components Added to the Dataset**

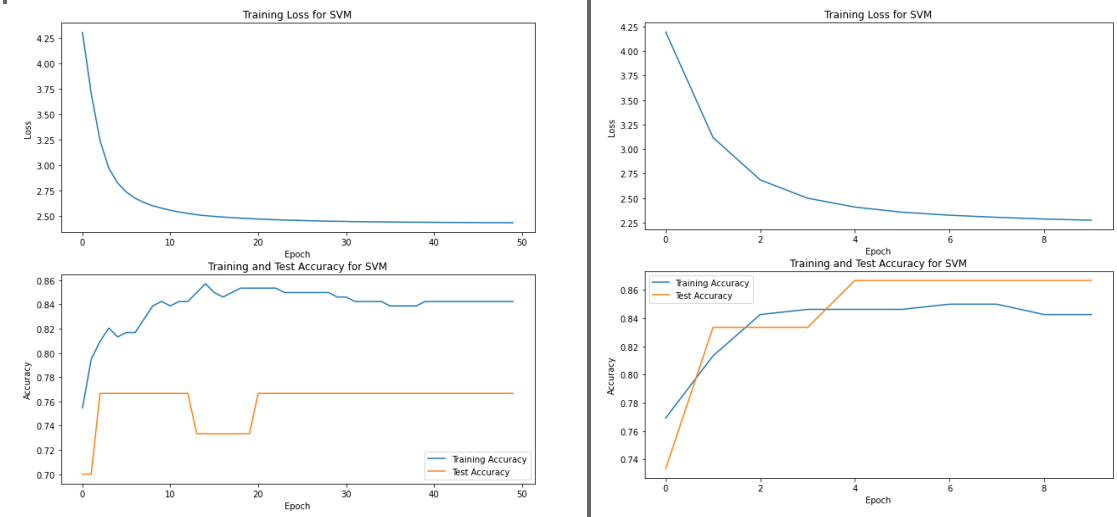


Figure 6: Result Graphs for SVM

6. Project Plan

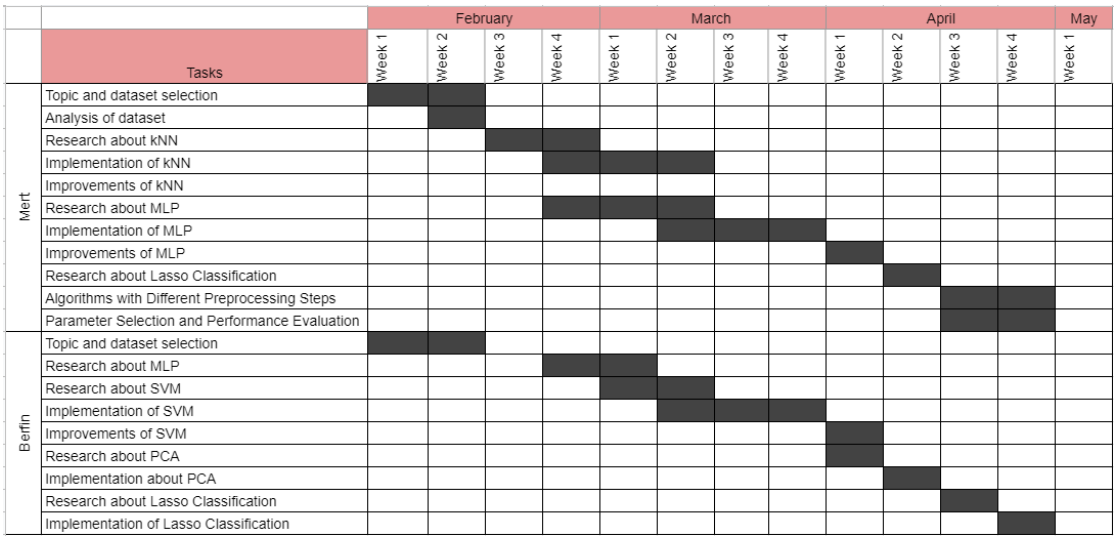


Figure 7: Gannt Chart

Until Phase I, analysis of the dataset, and the basic implementation of KNN, NN and SVM were done. After Phase I, we made small changes to implementations, searched optimal parameters via grid search and cross validation. We implemented LASSO classification and used it for both classification and analyzing features for selection. Later, we implemented PCA and used it as a feature engineering method for all models. Group members worked according to the tasks assigned in our Gantt chart.

7. Conclusion

Table 3: Result Summary of Algorithms (w/o PCA and w PCA)

	Lasso		**Neural Networks**		**KNN**		**SVM**	
	w/o PCA	**w PCA**	**w/o PCA**	**w PCA**	**w/o PCA**	**w PCA**	**w/o PCA**	**w PCA**
Accuracy	0.80	0.80	0.9	0.87	0.87	0.87	0.77	0.87
Precision	0.86	0.86	0.91	0.87	0.87	0.87	0.78	0.79
Recall	0.79	0.79	0.91	0.87	0.87	0.87	0.76	0.85
F1-score	0.78	0.78	0.9	0.87	0.87	0.87	0.76	0.81

Among all the model and dataset combinations, our neural network outperformed the others after its parameters were optimized through grid search, KNN and SVM came close to the neural network. Adding principal components only made a positive difference for our SVM. Lasso Classifier was used to observe the weights of the features which we interpreted as importance and compared to the correlations of the features with the target. It turned out that eliminating features was not optimal as Lasso did not zero out many features. Given that our neural network was the best performer, we can say that our classification problem benefits from using a model capable of fitting nonlinear and complex decision boundaries. Overall, this project was a valuable opportunity to rehearse and improve our understanding of basic machine learning models and observe how they perform in a particular scenario.

References

[1] J. Brownlee, “K-Nearest Neighbors for Machine Learning,” *Machine Learning Mastery*, 14-Aug-2020. [Online]. Available: https://machinelearningmastery.com/k-nearest-neighbors-for-machine-learning/. [Accessed: 02-Apr-2021].

[2] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*. New York: Springer, 2017.

[3] *UCI Machine Learning Repository: Heart Disease Data Set*. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Heart+Disease. [Accessed: 15-Feb-2021].

[4] V. Fonti, “Feature Selection using LASSO,” 30-Mar-2017. [Online]. Available: https://beta.vu.nl/nl/Images/werkstuk-fonti_tcm235-836234.pdf.

[5] DeepAI, “Perceptron,” *DeepAI*, 17-May-2019. [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/perceptron#:~:text=A Perceptron is an algorithm,a single-layer neural

network. [Accessed: 02-Apr-2021].

[6] M. Amine, “Neural Networks Overview,” *Medium*, 28-Apr-2020. [Online]. Available: https://towardsdatascience.com/neural-networks-overview-c3e6ab3e366b. [Accessed: 02-Apr-2021].

[7] *Lecture 9: SVM*. [Online]. Available: https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote09.html. [Accessed: 02-Apr-2021].

[8] *Support vector machines: The linearly separable case*. [Online]. Available: https://nlp.stanford.edu/IR-book/html/htmledition/support-vector-machines-the-linearly-separable-case-1.html#:~:text=The SVM in particular defines,the margin of the classifier.&text=Figure 15.1 shows the margin and support vectors for a sample problem. [Accessed: 02-Apr-2021].

APPENDIX

- **KnearestNeighbours.py**

```
"""
Authors: Berfin Kavşut -  21602459
        Mert Ertuğrul - 21703957
"""

import numpy as np
from scipy.stats import mode
import utilities, PCA
import pandas as pd
import matplotlib.pyplot as plt

#Distance Metrics

def euclidian_distance(x1,x2):
   return np.sqrt(np.sum((x1-x2)**2))

def cosine_distance(x1,x2):
          return 1 - np.dot(x1,x2) / (np.linalg.norm(x1) * np.linalg.norm(x2))

def manhattan_distance(x1, x2):
   return np.abs(x1 - x2).sum()


def KNN_classify(x_train, y_train , x_test, k, distance = "euclidian"):

   """
   K Nearest Neighbours algorithm computes the data point's distance to other
   data points accoridng to a given metric in the feauture space and selects the
   majority vote label of k nearest neighbours.

   x_train: training set features
   y_train: training set labels
   x_test: test set features
```

```
    k: number of nearest neighbors to vote for label
    distance: the distance metric to be used
    """

    result_labels = []

    #Loop through the datapoints to be classified
    for item in x_test:

        #distances to training data points
        distances = []

        for i in range(len(x_train)):
            #choose distance metric
            if distance == "euclidian":
              new_dist = euclidian_distance(np.array(x_train[i,:]) , item)
            elif distance == "cosine":
              new_dist = cosine_distance(np.array(x_train[i,:]) , item)
            else:
              new_dist = manhattan_distance(np.array(x_train[i,:]) , item)
            distances.append(new_dist)

        #sorts distances and returns their original indices, takes the last k distances
        indices = np.argsort( np.array(distances) )[:k]
        labels = y_train[indices]

        #most common label selected
        label = mode(labels)
        label = label.mode[0]
        result_labels.append(label)

    return result_labels


#standalone main function to demonstrate the model's k-fold cross validation results on graphs
def main_knn_grid_search(add_PCA = True):
    np.random.seed(1)

    #reading the data into a pandas dataframe and converting to numpy array
    data_heart = pd.read_csv('./heart.csv')

    D = data_heart.values

    #train test split
    D_train, D_test = utilities.split_train_test(D)

    #standardizing and shuffling the data
    standardizer = utilities.Standardizer()
    #train data

    D_train = standardizer.standardize_data( D_train )
```

```
    #test data - statistical proeprties of test data do not leak into standardization parameters
    D_test = standardizer.standardize_data( D_test, testing=True )

    #---------- adding PCA
    if add_PCA:
        pca_obj = PCA.PCA_maker(X= D_train[:, :-1], n_components=8)
        PCA_features_train = pca_obj.apply_pca(X=D_train[:, :-1], display=False)

        D_train = np.hstack( ( D_train[:, :-1], PCA_features_train, D_train[:, -1].reshape(D_train.shape[0],1)
) )
        #print(D_train.shape)

        PCA_features_test = pca_obj.apply_pca(X=D_test[:, :-1], display=False)
        D_test = np.hstack( ( D_test[:, :-1], PCA_features_test, D_test[:, -1].reshape(D_test.shape[0],1) ) )

    #number of folds for k fold cross validation
    k_fold=5


    #creating the individual folds and seperating the label vector
    X_folds,Y_folds = utilities.k_fold_split( D_train, k_fold )

    #range of k values to be tested for k nearest neighbors
    K_values = [3,5,7,9,11,13,15,19,21,23,25,50,70,100,125,150,175,200,225,250]
    #different distance types to be tested
    distances = ["euclidian", "cosine", "manhattan"]


    #keeps accuracy values for each distance metric
    accuracies_per_dist = []

    #preparing plot layout
    fig = plt.figure(figsize=(10,10))

    ax = fig.add_subplot(2, 1, 1)
    ax.set_xlabel('K value')
    ax.set_ylabel('Cross Validation Accuracy')
    ax.set_ylim([0, 1])

    ax2 = fig.add_subplot(2, 1, 2)
    ax2.set_xlabel('K value')
    ax2.set_ylabel('Cross Validation Accuracy')
    ax2.set_title('Impact of k and Distance Type on Average Accuracy ')
    ax2.set_ylim([0, 1])

    max_params= ()
    max_accuracy = 0

    #for each distance metric
    for dist_index, dist in enumerate(distances):
```

```
        #print("For distance type: "+dist)

        #keeps accuracy values for each run
        accuracy_data_k = np.zeros( (k_fold,len(K_values) ))

        #for each of the folds being used for validation
        for i in range(k_fold):
          #taking all folds other than the ith one for training
          X_train = np.concatenate( X_folds[:i] + X_folds[i+1:] , axis=0 )
          Y_train = np.concatenate( Y_folds[:i] + Y_folds[i+1:] , axis=0 )
          #ith fold is used for validation
          X_val = X_folds[i]
          Y_val = Y_folds[i]

          for k_index, k in enumerate(K_values):

              labels = KNN_classify(X_train, Y_train , X_val, k, dist)
              #average accuracy accross the validation fold
              accuracy = np.sum(labels == Y_val) / Y_val.shape[0]

              accuracy_data_k[i][ k_index] = accuracy


        #averaging accross folds
        avg_accuracy =  np.mean(accuracy_data_k, axis=0)

        #looking for the maximium accuracy
        j = np.argmax(avg_accuracy)
        if avg_accuracy[j]> max_accuracy:
            max_accuracy = avg_accuracy[j]
            max_params = (dist_index,j)

        accuracies_per_dist.append(accuracy_data_k)
        ax2.plot(K_values, avg_accuracy, label= dist)

    print("Maximum cv accuracy of " + str(max_accuracy) + " found for
k="+str(K_values[max_params[1]])+" distance:"+distances[max_params[0]])

    for i in range(k_fold):
        ax.plot(K_values, accuracies_per_dist[ max_params[0] ][i] )
    ax.plot(K_values, avg_accuracy, label= "average", linewidth=5.0)

    ax.set_title('k (knn parameter) vs CV Accuracy Graph for K Folds,
Distance:'+distances[max_params[0]])
    ax.legend()
    ax2.legend()
    plt.show()

    #fig.savefig('KNN_Plots_2.png')

    #----- TESTING ----------------------------
```

```
    print("Now we test the model using the best parameter set.")

    X_train = D_train[:, :-1]
    Y_train = D_train[:, -1]

    X_test = D_test[:, :-1]
    Y_test = D_test[:, -1]


    #testing
    Y_predicted = KNN_classify(X_train, Y_train , X_test, K_values[max_params[1]],
distances[max_params[0]])


    print("---- KNN Test Results ----")
    utilities.get_metrics(Y_test, Y_predicted, print_metrics=True)


if __name__ == '__main__':
  main_knn_grid_search()

  print("---Without adding PCA---")
  main_knn_grid_search(add_PCA = False)
  print("---With PCA---")
  main_knn_grid_search(add_PCA = True)
```

- **Lasso.py**

```
"""
Authors: Berfin Kavşut -  21602459
        Mert Ertuğrul - 21703957
"""

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

from sklearn.utils import shuffle

class Lasso():
    def __init__(self, l1_lambda=0.1):
        self.l1_lambda = l1_lambda
        self.w = None
        #no bias term, since data is expected to be centered

    def fit(self, X, Y, random_seed=1, num_epochs=3, learning_rate=0.01, show_graph=True, X_val =
None, Y_val = None):
```

```
sample_no = X.shape[0]
input_dim = X.shape[1]

# initializing coefficients of hyperplane
np.random.seed(random_seed)
self.w = np.random.randn(input_dim, 1) * 0.1

average_costs = []
accuracy_list = []
weight_list = np.zeros((input_dim,num_epochs))
if Y_val is not None:
    val_accuracy_list = []

for epoch in range(num_epochs):

    X,Y = shuffle(X,Y)  # stochastic gradient descent
    cost = 0
    for i in range(sample_no):
        x = X[i, :]
        x = np.reshape(x, (1, input_dim))
        y = Y[i]
        # update coefficients
        self.w = self.gradient_descent(self.w, x, y, learning_rate)
        cost += self.calc_cost(self.w, x, y)

    cost = cost / sample_no

    cost = np.asarray(cost)
    cost = np.squeeze(cost)

    y_preds = self.predict(X)

    accuracy =  self.get_accuracy(Y, y_preds)

    if Y_val is not None:
        y_val_preds = self.predict(X_val)

        val_accuracy =  self.get_accuracy(Y_val, y_val_preds)
        val_accuracy_list.append(val_accuracy)

    average_costs.append(cost)
    accuracy_list.append(accuracy)
    weight_list[:,epoch] = self.w

if show_graph:
    x_axis = np.arange(num_epochs)

    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(2, 1, 1)
    ax.plot(x_axis, average_costs)
```

```
        ax.set_xlabel('Epoch')
        ax.set_ylabel('Loss')

        ax2 = fig.add_subplot(2, 1, 2)
        ax2.plot(x_axis, accuracy_list, label = "Training Accuracy")
        ax2.set_xlabel('Epoch')
        ax2.set_ylabel('Accuracy')
        ax2.set_ylim((0.5,1))

        if Y_val is not None:
            ax2.set_ylabel('Accuracy')
            ax2.plot(x_axis, val_accuracy_list, label = "Test Accuracy")
            ax2.legend()
            ax2.set_title('Training and Test Accuracy for Lasso')
        else:
            ax2.set_title('Training Accuracy for Lasso')

        ax.set_title('Training Loss for Lasso')

        #plt.savefig('lasso_training_graphs.png')
        plt.show()
    return self.w, weight_list, average_costs, accuracy_list

def calc_cost(self, w, x, y):

    # calculate rss
    rss = (1/2) * np.sum( np.square( y - (np.dot(x, w) ) ))

    # calculate regularization term
    reg_term = self.l1_lambda * np.sum(np.abs(w))

    # calculate cost
    cost = rss + reg_term
    return cost

# returns average accuracy given real and predicted labels
def get_accuracy(self, Y_real, Y_predicted):

    Y_labelled = (Y_predicted >= 0)
    Y_labelled = np.where(Y_labelled == 0, -1, Y_labelled)

    return ( Y_labelled == Y_real).mean()


def gradient_descent(self, w, x, y, learning_rate):

    x = np.squeeze(x)
    w = np.squeeze(w)

    n = w.shape[0]
    dw = np.zeros([1,n])
```

```
        dw = np.squeeze(dw)

        # gradient of RSS+L1 loss
        for i in range(n):
            rss_der = -1 * x[i] * ( y-np.dot( x,w.T ) )
            if w[i] > 0:
                dw[i] = rss_der + self.l1_lambda
            else:
                dw[i] = rss_der - self.l1_lambda

        # update weights
        w = w - learning_rate * dw
        return w

    def predict(self,X):

        Y_pred = np.dot(X,self.w)
        return Y_pred
```

- **lasso_classifier_cv.py**

```
"""
Authors: Berfin Kavşut -  21602459
         Mert Ertuğrul - 21703957
"""


import Lasso, utilities, PCA

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

"""
Dataset is seperated as 90% training and cross validation set
                        10% test set

This class carries out grid search for the parameters of Lasso using cross-validation.
Afterwards, the best performing parameter set is used to train on the model on the entire training set
and test accuracy is obtained using the test set.

"""
feature_names = [ "age",
  "sex",
  "cp",
  "trestbps",
  "chol",
  "fbs",
  "restecg",
  "thalach",
```

```
  "exang",
  "oldpeak",
  "slope",
  "ca",
  "thal",
  "PC1","PC2","PC3","PC4","PC5","PC6","PC7","PC8"]

def main_lasso_classifier_cv(add_PCA = True, display_search_logs=False):

    np.random.seed(1)

    #reading the data into a pandas dataframe and converting to numpy array
    data_heart = pd.read_csv('./heart.csv')
    D = data_heart.values

    #train test split
    D_train, D_test = utilities.split_train_test(D)

    #standardizing and shuffling the data
    standardizer = utilities.Standardizer()

    #cross validation and training data
    D_train = standardizer.standardize_data( D_train )
    #test data - statistical proeprties of test data do not leak into standardization parameters
    D_test = standardizer.standardize_data( D_test, testing=True )

    X_test = D_test[:, :-1]
    Y_test = D_test[:, -1]
    Y_test = np.where(Y_test == 0, -1, Y_test)

    #---------- adding PCA
    if add_PCA:
        pca_obj = PCA.PCA_maker(X= D_train[:, :-1], n_components=8)
        PCA_features_train = pca_obj.apply_pca(X=D_train[:, :-1], display=False)

        D_train = np.hstack( ( D_train[:, :-1], PCA_features_train, D_train[:, -1].reshape(D_train.shape[0],1)
) )
        #print(D_train.shape)

        PCA_features_test = pca_obj.apply_pca(X=X_test, display=False)
        X_test = np.hstack( ( X_test, PCA_features_test) )


    #------------------- CROSS VALIDATION ----------------------------------------
    k_fold = 5

    #Preparing the training data for cross validation
    X_folds,Y_folds = utilities.k_fold_split( D_train, k_fold )

    #parameter grid
    num_epochs_list = [1,2,3,4,5,10]
```

```
learning_rate_list = [0.0001, 0.001, 0.005,0.01, 0.05, 0.1,]
lambda_list = [0,0.001,0.01,0.1]

best_params = ()
best_accuracy = 0

for num_epochs in num_epochs_list:
    for learning_rate in learning_rate_list:
        for l1_lambda in lambda_list:

            cv_accuracy = []
            cv_weights = []

            for i in range(k_fold):

                #creating training data by omitting the ith fold
                X_train = np.concatenate( X_folds[:i] + X_folds[i+1:] , axis=0 )
                Y_train = np.concatenate( Y_folds[:i] + Y_folds[i+1:] , axis=0 )
                # change labels of 0 to -1, negative class
                Y_train = np.where(Y_train == 0, -1, Y_train)

                #ith fold is the test data
                X_val = X_folds[i]
                Y_val = Y_folds[i]

                # change labels of 0 to -1, negative class
                Y_val = np.where(Y_val == 0, -1, Y_val)

                # Lasso model
                lasso_model = Lasso.Lasso(l1_lambda=l1_lambda)

                weights,_,_,_ = lasso_model.fit( X_train, Y_train, num_epochs=num_epochs,
                                    learning_rate=learning_rate, show_graph=False )
                #store final weights of features
                cv_weights.append(weights)

                #validation
                Y_pred = lasso_model.predict(X_val)
                accuracy = lasso_model.get_accuracy(Y_val, Y_pred)
                cv_accuracy.append(accuracy)


                #overall average test accuracy
                avg_cv_accuracy = np.mean(cv_accuracy)

                if avg_cv_accuracy > best_accuracy:

                    best_accuracy = avg_cv_accuracy
                    best_params = (num_epochs,learning_rate,l1_lambda)

                    if display_search_logs:
```

```
                    print("New Best Accuracy of:",best_accuracy)
                    print("New Best Params:    Epochs:",num_epochs,"  LR:",learning_rate,"
LAMB:",l1_lambda)
                    print("----------------------------------------------------------------------")


    print("Final CV Accuracy: ",best_accuracy)
    print("Final Params:    Epochs:",best_params[0],"  LR:",best_params[1],"  LAMB:",best_params[2])

    print("Now we test the best parameters on test set:")
    X_train = D_train[:, :-1]
    Y_train = D_train[:, -1]
    Y_train = np.where(Y_train == 0, -1, Y_train)

    lasso_model = Lasso.Lasso( l1_lambda = best_params[2] )


    # Training the model
    weights,weight_arr,_,_  = lasso_model.fit( X_train, Y_train, num_epochs=best_params[0],
                                learning_rate=best_params[1], show_graph=True, X_val = X_test,  Y_val =
Y_test )

    # Displaying how the weights of the features changed over the epochs
    fig = plt.figure(figsize=(10,10))

    x_axis = np.arange(best_params[0])

    for i in range(weight_arr.shape[0]):
        plt.plot(x_axis, np.abs(weight_arr[i]), label= feature_names[i] )

    plt.title("Weight Change vs Epoch")
    plt.xlabel("Epoch")
    plt.ylabel("Absoulte Value of Weight")
    plt.legend(loc=(1.04,0.5))
    plt.show()

    #testing
    Y_pred = lasso_model.predict(X_test)

    print("---- Lasso Test Results ----")
    utilities.get_metrics(Y_test, np.squeeze(Y_pred), print_metrics=True)

    print("Final Weights:\n",weights)

if __name__ == '__main__':

  print("---Without adding PCA---")
  main_lasso_classifier_cv(add_PCA = False)
  print("---With PCA---")
  main_lasso_classifier_cv(add_PCA = True)
```

- **lasso_feature_selection.py**

```
"""
Authors: Berfin Kavşut -  21602459
        Mert Ertuğrul - 21703957
"""


import Lasso, utilities, PCA

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

feature_names = [ "age",
  "sex",
  "cp",
  "trestbps",
  "chol",
  "fbs",
  "restecg",
  "thalach",
  "exang",
  "oldpeak",
  "slope",
  "ca",
  "thal",
  "PC1","PC2","PC3","PC4","PC5","PC6","PC7","PC8"]

def main_lasso_feature_selection(add_PCA = True, display_search_logs=False):
    np.random.seed(1)
    num_epochs=10
    learning_rate=0.01
    l1_lambda = 0.05

    #reading the data into a pandas dataframe and converting to numpy array
    data_heart = pd.read_csv('./heart.csv')
    D = data_heart.values

    #train test split
    D_train, D_test = utilities.split_train_test(D)

    #standardizing and shuffling the data
    standardizer = utilities.Standardizer()

    #cross validation and training data
    D_train = standardizer.standardize_data( D_train )

    X_train = D_train[:, :-1]
```

```
Y_train = D_train[:, -1]
Y_train = np.where(Y_train == 0, -1, Y_train)

   #----------- adding PCA
if add_PCA:
   pca_obj = PCA.PCA_maker(X=X_train, n_components=8)
   PCA_features_train = pca_obj.apply_pca(X=X_train, display=False)

   X_train = np.hstack( ( X_train, PCA_features_train ) )


# Lasso model
lasso_model = Lasso.Lasso( l1_lambda = l1_lambda)
final_weights, weight_arr,_,_  = lasso_model.fit( X_train, Y_train, num_epochs=num_epochs,
                                learning_rate=learning_rate, show_graph=False )


x_axis = np.arange(num_epochs)

for i in range(weight_arr.shape[0]):
   plt.plot(x_axis, np.abs(weight_arr[i]), label= feature_names[i] )

plt.title("Weight Change vs Epoch")
plt.xlabel("Epoch")
plt.ylabel("Absoulte Value of Weight")
plt.legend(loc=(1.04,0.5))
plt.show()


#-------------------------------------------------------------------------------------
lambda_list = [0,0.0001,0.0005,0.001,0.005,0.01,0.05,0.1,0.5]

weights_vs_lambda = np.zeros( (X_train.shape[1], len(lambda_list)) )
for index, lamb in enumerate(lambda_list):
   # Lasso model
   lasso_model = Lasso.Lasso( l1_lambda =lamb)
   final_weights, weight_arr,_,_  = lasso_model.fit( X_train, Y_train, num_epochs=num_epochs,
                                learning_rate=learning_rate, show_graph=False )

   weights_vs_lambda[:,index] = final_weights

#Showing the plot for different lambdas
fig = plt.figure()
for i in range(weight_arr.shape[0]):
   plt.plot(lambda_list, np.abs(weights_vs_lambda[i]), label= feature_names[i] )

plt.title("Weight Change vs Lambda")
plt.xlabel("Lambda")
plt.ylabel("Absoulte Value of Weight")
plt.legend(loc=(1.04,0.5))
plt.show()
```

```
if __name__ == '__main__':

  print("---Without adding PCA---")
  main_lasso_feature_selection(add_PCA = False)
  print("---With PCA---")
  main_lasso_feature_selection(add_PCA = True)
```

- **NeuralNetwork.py**

```
"""
Authors: Berfin Kavşut -  21602459
        Mert Ertuğrul - 21703957
"""

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


class networkLayer:

    """
    This class represents a signel layer of a multi layer perceptron.
    The layer stores:
        -W: weight matrix parameter
        -b: bias parameter
        -Z: output of the last iteration - used for back propagation
        -d_W: gradient of W
        -d_b: gradient of b

    """

    def __init__(self, in_dim, out_dim, activation_func):

      self.activation_func = activation_func

      #initializing parameters of layer

      #we keep the random seed constant to make our results repeatable by 3rd parties
      self.W = np.random.randn(out_dim, in_dim) * 0.1
      self.b = np.random.randn(out_dim, 1) * 0.1

      #storing output for back prop
      self.Z = 0
      self.input_X = 0

      #each layer stores its gradient
      self.d_W = 0
```

```
  self.d_b = 0

#applies forward propogation to the layer and keeps result
def forward_prop(self, input_X, training=True):

  if training:
     self.input_X = input_X
     self.Z = np.dot(self.W, input_X) + self.b

     #activation function
     if self.activation_func == "sigmoid":
       return self.sigmoid(self.Z)
     elif self.activation_func == "relu":
       return self.relu(self.Z)
  else:

     Z = np.dot(self.W, input_X) + self.b

     #activation function
     if self.activation_func == "sigmoid":
       return self.sigmoid(Z)
     elif self.activation_func == "relu":
       return self.relu(Z)

#applies back propogation, stores the gradients of the parameters,
#returns gradient of input to be used for the preceding layer
def backward_prop(self, d_out):

  #gradient of Z
  if self.activation_func == "sigmoid":
    d_Z = d_out * self.sigmoid(self.Z) * (1 - self.sigmoid(self.Z))

  elif self.activation_func == "relu":
    d_Z = d_out.copy()
    d_Z[self.Z <= 0] = 0

  #gradient of params
  self.d_W = (1/self.input_X.shape[1]) * np.dot(d_Z, self.input_X.T)
  self.d_b = (1/self.input_X.shape[1]) * np.sum(d_Z, axis=1, keepdims=True)
  #gradient of input
  d_in = np.dot(self.W.T, d_Z)

  #clearing stored input for next epoch
  self.input_X = 0

  return d_in

#updates paraemeters based on gradients and learning rate
def update(self, learning_rate):
  self.W -= self.d_W * learning_rate
  self.b -= self.d_b * learning_rate
```

```
    #relu activation function used for hidden layers
    def relu(self, X):
        return np.maximum(0,X)

    #sigmoid activation function used for last layer
    def sigmoid(self, X):
      return 1/(1+np.exp(-X))



class NeuralNetwork:

    """
    This class represents a Multi Layer Perceptron. New layers are added,
    forward and backward propogation, training and testing functionalities
    are provided.

    """
    epsilon = 1e-6

    def __init__(self, learning_rate = 0.01, threshold = 0.5):
      self.learning_rate = learning_rate
      self.threshold = threshold
      self.layers = []

    def addLayer(self, input_dim, output_dim, activation):
      self.layers.append(networkLayer(input_dim, output_dim, activation))

    def forward_prop(self, X, training=True):

      Z_temp = X
      for layer_num in range(len(self.layers)):
        Z_temp = self.layers[layer_num].forward_prop(Z_temp, training=training)
      return Z_temp


    def form_architecture(self, input_dim = 13, hidden_layers = 2):
        #forming architecture
        if hidden_layers==0:
            self.addLayer(input_dim, 1, "sigmoid")
        else:
            if hidden_layers==1:
                self.addLayer(input_dim, 8, "relu")
            else:
                self.addLayer(input_dim, 16, "relu")
                for i in range(hidden_layers-2):
                    self.addLayer(16, 16, "relu")
                self.addLayer(16, 8, "relu")

            self.addLayer(8, 1, "sigmoid")
```

```
    def backward_prop(self, Y_real, Y_predicted):

      #derivative of logit loss to be sent back
      d_in = - (np.divide(Y_real, Y_predicted + self.epsilon) - np.divide(1 - Y_real, 1 - Y_predicted +
self.epsilon ))

      for layer_num in reversed( range(len(self.layers) ) ):
        #store output gradient (input gradient of the following layer)
        d_out = d_in
        #get input gradient
        d_in = self.layers[layer_num].backward_prop( d_out )

    def update_params(self):
      for layer_num in range(len(self.layers)):
        self.layers[layer_num].update(self.learning_rate)


    # returns the logit loss given real and predicted labels
    def logit_loss(self, Y_real, Y_predicted):
      loss = (-1 / Y_predicted.shape[1] )*(np.dot(Y_real, np.log(Y_predicted + self.epsilon).T) + np.dot(1 -
Y_real, np.log(1 - Y_predicted  + self.epsilon).T))
      return np.squeeze(loss)

    # returns average accuracy given real and predicted labels
    def get_accuracy(self, Y_real, Y_predicted):
      return ( (Y_predicted >= self.threshold)== Y_real).mean()


    def train(self, X, Y_real,num_epochs, show_graph = True, mini_batch_size=0, X_val = None, Y_val =
None):

      loss_list = []
      accuracy_list = []

      if Y_val is not None:
         val_accuracy_list = []

      if mini_batch_size != 0:
        batch_num = math.floor( X.shape[0]/mini_batch_size )

        X_batches = np.array_split(X, batch_num)
        Y_batches = np.array_split(Y_real, batch_num)
      else:
        X_batches = [X]
        Y_batches = [Y_real]

      for i in range(num_epochs):

        for b in range(len(X_batches)):
```

```
        Y_predicted = self.forward_prop(X_batches[b].T)

        self.backward_prop(Y_batches[b], Y_predicted)
        self.update_params()


    accuracy,Y_predicted = self.test(X , Y_real)
    loss = self.logit_loss(Y_real, Y_predicted)
    #print( "Loss: " + str(loss) )
    loss_list.append( loss )

    #print( "Accuracy: " + str(accuracy) )
    accuracy_list.append( accuracy )

    if Y_val is not None:
        val_acc,_ = self.test(X_val , Y_val)
        val_accuracy_list.append(val_acc)



#option to display the training loss and accuracy graph

if show_graph:

    x_axis = np.arange(num_epochs)
    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(2, 1, 1)
    ax.plot(x_axis, loss_list)
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Training Loss')
    ax.set_title('Training Loss for Neural Network')

    ax2 = fig.add_subplot(2, 1, 2)
    ax2.plot(x_axis, accuracy_list, label = "Training Accuracy")
    #We are using the test acuracy only for display,
    #it does not influence training so we do not call it Validation Accuracy here.
    ax2.set_xlabel('Epoch')
    ax2.set_ylim((0.5,1))

    if Y_val is not None:
        ax2.set_ylabel('Accuracy')
        ax2.plot(x_axis, val_accuracy_list, label = "Test Accuracy")
        ax2.set_title('Training and Test Accuracy for Neural Network')
        ax2.legend()
    else:
        ax2.set_ylabel('Training Accuracy')
        ax2.set_title('Training Accuracy for Neural Network')

    plt.show()
```

```
        return loss_list, accuracy_list

    #given a test set, returns the accuracy and predicted labels
    def test(self, X, Y_real):
        Y_predicted = self.forward_prop(X.T, training=False)
        return self.get_accuracy(Y_real, Y_predicted), Y_predicted

#standalone main function to demonstrate the model's k-fold cross validation results on graphs
def main_cv_grapher():

    np.random.seed(1)

    #reading the data into a pandas dataframe and converting to numpy array
    data_heart = pd.read_csv('./heart.csv')

    #standardizing and shuffling the data
    standardizer = utilities.Standardizer()

    D = standardizer.standardize_data( data_heart.values )

    #number of folds for k fold cross validation
    k=10
    num_epochs = 200
    learning_rate = 0.1

    X_folds,Y_folds = utilities.k_fold_split( D, k )

    loss_data = np.zeros( (10,num_epochs) )
    accuracy_data = np.zeros( (10,num_epochs) )

    val_accuracy = []

    x_axis = np.arange(num_epochs)

    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(2, 1, 1)
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Loss')
    ax.set_ylim([0, 1])

    ax2 = fig.add_subplot(2, 1, 2)
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Accuracy')
    ax2.set_ylim([0, 1])

    for i in range(10):

        #creating training data by omitting the ith fold
        X_train = np.concatenate( X_folds[:i] + X_folds[i+1:] , axis=0 )
        Y_train = np.concatenate( Y_folds[:i] + Y_folds[i+1:] , axis=0 )
```

```
    #ith fold is the validation data
    X_val = X_folds[i]
    Y_val = Y_folds[i]

    newNetwork = NeuralNetwork(learning_rate)
    #forming architecture
    newNetwork.addLayer(13, 16, "relu")
    newNetwork.addLayer(16, 8, "relu")
    newNetwork.addLayer(8, 1, "sigmoid")
   #training the network
    loss_series, train_accuracy_series = newNetwork.train(X_train,Y_train,num_epochs, False)

    #keeping track of the logit loss and train set accuracy series to
    #plot against epochs
    loss_data[i] = np.array(loss_series)
    accuracy_data[i] = np.array(train_accuracy_series)

    #testing with ith fold
    avg_val_result,val_result = newNetwork.test(X_val,Y_val)
    print("Iteration " + str(i) + " validation accuracy: " + str(avg_val_result) )
    val_accuracy.append( avg_val_result )

    ax.plot(x_axis, loss_series )
    ax2.plot(x_axis, train_accuracy_series )

  avg_accuracy =  np.mean(accuracy_data, axis=0)
  avg_loss =  np.mean(loss_data, axis=0)

  #average values accross the 10 folds
  ax.plot(x_axis, avg_loss, label= "average loss", linewidth=3.0)
  ax2.plot(x_axis, avg_accuracy, label= "average accuracy", linewidth=3.0)


  # Set a title of the current axes.
  ax.set_title('Neural Network Training Loss Graph for 10-fold Cross Validation')
  ax2.set_title('Neural Network Training Accuracy Graph for 10-fold Cross Validation')
  ax.legend()
  ax2.legend()
  plt.show()

  #overall average test accuracy
  avg_val_accuracy = np.mean(val_accuracy)

  print("Average validation accuracy: "+str(avg_val_accuracy))



if __name__ == '__main__':
  main_cv_grapher()
```

- **NN_grid_search.py**

```
"""
Authors: Berfin Kavşut -  21602459
        Mert Ertuğrul - 21703957
"""

import NeuralNetwork, utilities, PCA

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


def grid_search_neural_network(add_PCA = True, display_search_logs=False):
    np.random.seed(1)

    #reading the data into a pandas dataframe and converting to numpy array
    data_heart = pd.read_csv('./heart.csv')

    D = data_heart.values

    #train test split
    D_train, D_test = utilities.split_train_test(D)

    #standardizing and shuffling the data
    standardizer = utilities.Standardizer()
    #train data

    D_train = standardizer.standardize_data( D_train )
    #test data - statistical proeprties of test data do not leak into standardization parameters
    D_test = standardizer.standardize_data( D_test, testing=True )

    X_test = D_test[:, :-1]
    Y_test = D_test[:, -1]


    #----------- adding PCA
    if add_PCA:
        pca_obj = PCA.PCA_maker(X= D_train[:, :-1], n_components=8)
        PCA_features_train = pca_obj.apply_pca(X=D_train[:, :-1], display=False)

        D_train = np.hstack( ( D_train[:, :-1], PCA_features_train, D_train[:, -1].reshape(D_train.shape[0],1)
) )
        #print(D_train.shape)

        PCA_features_test = pca_obj.apply_pca(X=X_test, display=False)
        X_test = np.hstack( ( X_test, PCA_features_test) )

    #number of folds for k fold cross validation
```

```
k_fold=5

#-------parameter grid for grid search-----------------------
num_epochs_list = [2,5,10,25]
learning_rate_list = [0.01,0.1, 0.3,0.5,0.7]

   #batch_size = 0 -> batch grad decent
   #batch_size = 1 -> stochastic grad decent
   #batch_size = other -> mini batch grad decent
mini_batch_size_list = [0,1,5,10,20,50]
num_hidden_layers_list = [0,1,2,3]


#Preparing the training data for cross validation
X_folds,Y_folds = utilities.k_fold_split( D_train, k_fold )

best_params = ()
best_accuracy = 0

for num_epochs in num_epochs_list:
   for learning_rate in learning_rate_list:
      for  hidden_layers in num_hidden_layers_list:
         for mini_batch_size in mini_batch_size_list:

            cv_accuracy = []


            for i in range(k_fold):

               #creating training data by omitting the ith fold
               X_train = np.concatenate( X_folds[:i] + X_folds[i+1:] , axis=0 )
               Y_train = np.concatenate( Y_folds[:i] + Y_folds[i+1:] , axis=0 )

               #ith fold is the test data
               X_val = X_folds[i]
               Y_val = Y_folds[i]

               newNetwork = NeuralNetwork.NeuralNetwork(learning_rate)

               newNetwork.form_architecture(input_dim = X_train.shape[1], hidden_layers =
hidden_layers)

               #training the network
               newNetwork.train(X_train,Y_train,num_epochs, False, mini_batch_size)

               #validating with ith fold
               val_accuracy,_ = newNetwork.test(X_val,Y_val)
               cv_accuracy.append( val_accuracy )

            #overall average test accuracy
            avg_cv_accuracy = np.mean(cv_accuracy)
```

```
                if avg_cv_accuracy > best_accuracy:
                    best_accuracy = avg_cv_accuracy
                    best_params = (num_epochs,learning_rate,hidden_layers,mini_batch_size)

                    if display_search_logs:
                        print("New Best Accuracy of:",best_accuracy)
                        print("New Best Params:    Epochs:",num_epochs,"  LR:",learning_rate,"
HL:",hidden_layers,"  MBS:",mini_batch_size)
                        print("-------------------------------------------------------------------------")



    print("\n**********************************************************\n")
    print("Final CV Accuracy: ",best_accuracy)
    print("Final Params:    Epochs:",best_params[0],"  LR:",best_params[1],"  HL:",best_params[2],"
MBS:",best_params[3])

    print("Now we test the model using the best parameter set.")

    X_train = D_train[:, :-1]
    Y_train = D_train[:, -1]

    newNetwork = NeuralNetwork.NeuralNetwork(learning_rate=best_params[1])

    newNetwork.form_architecture(input_dim = X_train.shape[1], hidden_layers = best_params[2])


    #training the network and displaying graphs
    newNetwork.train(X_train,Y_train,best_params[0], True, best_params[3], X_val = X_test,  Y_val =
Y_test)

    #testing
    test_accuracy, Y_predicted = newNetwork.test(X_test,Y_test)

    print("---- Neural Network Test Results ----")
    utilities.get_metrics(Y_test, np.squeeze(Y_predicted), print_metrics=True)



if __name__ == '__main__':

  print("---Without adding PCA---")
  grid_search_neural_network(add_PCA = False)
  print("---With PCA---")
  grid_search_neural_network(add_PCA = True)
```

- **PCA.py**

```
import utilities
```

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math
from sklearn.preprocessing import StandardScaler


class PCA_maker:

    def __init__(self, X, n_components):

        self.k = n_components

        # normalizing the features wrt training set
        self.standardScaler = StandardScaler().fit(X)
        X = self.standardScaler.transform(X)

        n = X.shape[0] #sample number

        # p x p, p is feature number
            sigma = (1/2) * np.dot(X.T, X)

        #each column of eigen_vectors is u, eigen_values are lambdas
            eigen_values, eigen_vectors = np.linalg.eig(sigma)

        #now, each row is eigenvector (for sorting)
            eigen_vectors = eigen_vectors.T

            #sort wrt eigenvalues
            sorted_indices = np.argsort(eigen_values)
            sorted_indices = np.flipud(sorted_indices)

            eigen_values = eigen_values[sorted_indices]
            eigen_vectors = eigen_vectors[sorted_indices]

        #each row is one eigenvector
            eigen_vectors = np.array(eigen_vectors)

        #saving eigenvectors/principal component loading vectors
        self.U = eigen_vectors.T

        #principal component scores, n x k
        #Z = np.dot( X, self.U[:,0:self.k] )

        #check eigendecomposition
        #D = np.diag(eigen_values)
        #error = np.sum(np.square(np.dot(U,D)-np.dot(sigma,U)))
        #print('MSE:', error)
```

```
def apply_pca(self, X, display=False):

    X = self.standardScaler.transform(X)
    n = X.shape[0] #sample number

    #principal component scores, n x k
    Z = np.dot( X, self.U[:,0:self.k] )

    if display == True:
        PVE_k_list = []
        PVE_m_list = []
        PVE_k = 0
        for m in range(self.k):
            total_variance = (1/n)*np.sum(np.square(X)) #13 directly since there are 13 features
            variance_explained = (1/n)*np.sum(np.square(Z[:,m]))

            PVE_m = variance_explained/total_variance
            PVE_m_list.append(PVE_m)
            PVE_k = PVE_k + PVE_m
            PVE_k_list.append(PVE_k)

        #print('PVE_k:', PVE_k_list)
        #print('PVE_m:', PVE_m_list)


        x_axis = np.arange(self.k)

        fig = plt.figure(figsize=(10,10))

        ax = fig.add_subplot(2, 1, 1)
        ax.plot(x_axis, PVE_m_list)
        ax.set_xlabel('Principal Component')
        ax.set_ylabel('Explained Variance')

        ax2 = fig.add_subplot(2, 1, 2)
        ax2.plot(x_axis, PVE_k_list)
        ax2.set_xlabel('Number of Principal Components')
        ax2.set_ylabel('Cumulative Explained Variance')

        ax.set_title('Explained Variance for Each Principal Component')
        ax2.set_title('Cumulative Explained Variance for PCA')

        fig.tight_layout(pad=3.0)

        plt.savefig('PCA.png')
        plt.show()

    return Z
```

```
def main_pca():

    data_heart = pd.read_csv('./heart.csv')

    data = data_heart.values

    # split features and label
    X = data[:, :-1]
    X = StandardScaler().fit_transform(X) # normalizing the features

    Y = data[:, -1]
    Y = np.reshape(Y,(Y.shape[0],1)) #to concatenate later

    #my model
    k = 13 #principal component analysis, dimension number

    pca_obj = PCA_maker(X=X, n_components=k)
    principal_component_scores = pca_obj.apply_pca(X=X, display=True)


    #-----Plotting the first two principal components against each other
    plt.figure()
    plt.figure(figsize=(10,10))
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=14)
    plt.xlabel('Principal Component - 1',fontsize=20)
    plt.ylabel('Principal Component - 2',fontsize=20)
    plt.title("Principal Component Analysis of Heart Disease Dataset",fontsize=20)
    targets = [0, 1] #benign, malignant
    colors = ['r', 'g']

    for target, color in zip(targets,colors):
        indicesToKeep = data_heart['target'] == target
        plt.scatter(principal_component_scores[indicesToKeep, 0],
                    principal_component_scores[indicesToKeep, 1], c = color, s = 50)

    plt.legend(targets,prop={'size': 15})


if __name__ == '__main__':
  main_pca()
```

- **SVM.py**

```
"""
Authors: Berfin Kavşut -  21602459
        Mert Ertuğrul - 21703957
"""

import numpy as np
```

```
import pandas as pd
import matplotlib.pyplot as plt
import utilities

from sklearn.utils import shuffle


class svm:
    def __init__(self, reg_param=1.0):
        # regularization parameter, C
        self.reg_param = reg_param
        self.w = None

    def fit(self, X, Y, random_seed=1, num_epochs=50, learning_rate=0.001, show_graph=True, X_val =
None, Y_val = None):

        # change labels of 0 to -1, negative class
        Y = np.where(Y == 0, -1, Y)

        sample_no = X.shape[0]
        input_dim = X.shape[1]

        # initializing coefficients of hyperplane
        np.random.seed(random_seed)
        self.w = np.random.randn(input_dim, 1) * 0.1

        average_costs = []
        accuracy_list = []
        if Y_val is not None:
            val_accuracy_list = []

        for epoch in range(num_epochs):

            X,Y = shuffle(X,Y)  # stochastic gradient descent
            cost = 0
            for i in range(sample_no):
                x = X[i, :]
                x = np.reshape(x, (1, input_dim))
                y = Y[i]
                # update coefficients
                self.w = self.gradient_descent(self.w, x, y, learning_rate)
                cost += self.calc_cost(self.w, x, y)

            cost = cost / sample_no

            cost = np.asarray(cost)
            cost = np.squeeze(cost)

            accuracy, y_preds = self.score(X, Y)

            if Y_val is not None:
```

```
            val_accuracy,_ = self.score(X_val,Y_val)
            val_accuracy_list.append(val_accuracy)

        average_costs.append(cost)
        accuracy_list.append(accuracy)

    if show_graph:
        x_axis = np.arange(num_epochs)

        fig = plt.figure(figsize=(10,10))
        ax = fig.add_subplot(2, 1, 1)
        ax.plot(x_axis, average_costs)
        ax.set_xlabel('Epoch')
        ax.set_ylabel('Loss')

        ax2 = fig.add_subplot(2, 1, 2)
        ax2.plot(x_axis, accuracy_list, label = "Training Accuracy")
        ax2.set_xlabel('Epoch')
        ax2.set_ylabel('Accuracy')

        if Y_val is not None:
            ax2.set_ylabel('Accuracy')
            ax2.plot(x_axis, val_accuracy_list, label = "Test Accuracy")
            ax2.legend()
            ax2.set_title('Training and Test Accuracy for SVM')
        else:
            ax2.set_title('Training Accuracy for SVM')

        ax.set_title('Training Loss for SVM')

        # plt.savefig('svm_accuracy_loss.png')
        plt.show()

def score(self, X, Y):
    # change labels of 0 to -1, negative class
    Y = np.where(Y == 0, -1, Y)

    # find score for given input X and Y
    sample_no = X.shape[0]
    y_preds = []
    count = 0

    # get rid of for loop later
    for i in range(sample_no):
        x = X[i, :]
        y_real = Y[i]

        output = np.dot(x, self.w)
        if output >= 0:
            y_pred = 1
        else:
```

```
                y_pred = -1
            y_preds.append(y_pred)

            if y_pred == y_real:
                count += 1

        accuracy = count / sample_no
        y_preds = np.array(y_preds)

        return accuracy, y_preds

    def calc_cost(self, w, x, y):
        # calculate hinge loss
        d = 1 - y * (np.dot(x, w))
        hinge_loss = max(d, 0)

        # calculate regularization term
        reg_term = 1 / 2 * np.dot(np.transpose(w), w)

        # calculate cost
        cost = reg_term + self.reg_param * hinge_loss
        return cost

    def gradient_descent(self, w, x, y, learning_rate):

        dw = 0
        # gradient of hinge loss
        if 1 - y * np.dot(x, w) <= 0:
            dw = w
        else:
            dw = w - self.reg_param * y * np.transpose(x)

        # update weights
        w = w - learning_rate * dw
        return w

def main():
    np.random.seed(1)

    data_heart = pd.read_csv('./heart.csv')

    # standardize data
    input = utilities.standardize_data(data_heart.values)

    # split features and label
    X = input[:, :-1]
    # print(X.shape)
    Y = input[:, -1]
    # print(Y.shape)

    # split training set and test set
```

```
    X_train = X[:int(X.shape[0] * 0.9)]
    # print(X_train.shape)
    X_test = X[int(X.shape[0] * 0.9):]
    # print(X_test.shape)
    Y_train = Y[:int(Y.shape[0] * 0.9)]
    # print(Y_train.shape)
    Y_test = Y[int(Y.shape[0] * 0.9):]
    # print(Y_test.shape)

    # change labels of 0 to -1, negative class
    Y_train = np.where(Y_train == 0, -1, Y_train)
    Y_test = np.where(Y_test == 0, -1, Y_test)

    # SVM model
    svm_model = svm()
    costs = svm_model.fit(X_train, Y_train)

    # accuracy on training set
    accuracy, y_preds = svm_model.score(X_train, Y_train)
    print('Training Accuracy:', accuracy)

    # accuracy on test set
    accuracy, y_preds = svm_model.score(X_test, Y_test)
    print('Test Accuracy:', accuracy)

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

- **SVM_grid_search.py**

```
"""
Authors: Berfin Kavşut -  21602459
         Mert Ertuğrul - 21703957
"""

import SVM, utilities, PCA

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


def grid_search_svm(add_PCA = True, display_search_logs=False):
    #reading the data into a pandas dataframe and converting to numpy array
    data_heart = pd.read_csv('./heart.csv')

    D = data_heart.values

    #train test split
    D_train, D_test = utilities.split_train_test(D)
```

```
    #standardizing and shuffling the data
    standardizer = utilities.Standardizer()
    #train data

    D_train = standardizer.standardize_data( D_train )
    #test data - statistical proeprties of test data do not leak into standardization parameters
    D_test = standardizer.standardize_data( D_test, testing=True )
    X_test = D_test[:, :-1]
    Y_test = D_test[:, -1]

    #----------- adding PCA
    if add_PCA:
        pca_obj = PCA.PCA_maker(X= D_train[:, :-1], n_components=8)
        PCA_features_train = pca_obj.apply_pca(X=D_train[:, :-1], display=False)

        D_train = np.hstack( ( D_train[:, :-1], PCA_features_train, D_train[:, -1].reshape(D_train.shape[0],1)
) )
        #print(D_train.shape)

        PCA_features_test = pca_obj.apply_pca(X=X_test, display=False)
        X_test = np.hstack( ( X_test, PCA_features_test) )

    #number of folds for k fold cross validation
    k_fold=5

    #-------parameter grid for grid search----------------------
    num_epochs_list = [10,25,50,75]
    learning_rate_list = [0.0001,0.001,0.01, 0.1]
    reg_param_list = [0.1, 0.5, 1, 2, 5]



    #Preparing the training data for cross validation
    X_folds,Y_folds = utilities.k_fold_split( D_train, k_fold )

    best_params = ()
    best_accuracy = 0

    for num_epochs in num_epochs_list:
        for learning_rate in learning_rate_list:
            for  reg_param in reg_param_list:

                cv_accuracy = []


                for i in range(k_fold):

                    #creating training data by omitting the ith fold
                    X_train = np.concatenate( X_folds[:i] + X_folds[i+1:] , axis=0 )
                    Y_train = np.concatenate( Y_folds[:i] + Y_folds[i+1:] , axis=0 )
```

```
                #ith fold is the test data
                X_val = X_folds[i]
                Y_val = Y_folds[i]

                svm_model = SVM.svm(reg_param=reg_param)

                #training the network
                svm_model.fit(X_train,Y_train,num_epochs=num_epochs,
                        learning_rate=learning_rate, show_graph=False)

                #validating with ith fold
                val_accuracy,_ = svm_model.score( X_val, Y_val)
                cv_accuracy.append( val_accuracy )

            #overall average test accuracy
            avg_cv_accuracy = np.mean(cv_accuracy)

            if avg_cv_accuracy > best_accuracy:
                best_accuracy = avg_cv_accuracy
                best_params = (num_epochs,learning_rate,reg_param)

                if display_search_logs:
                    print("New Best Accuracy of:",best_accuracy)
                    print("New Best Params:    Epochs:",num_epochs,"  LR:",learning_rate,"
REG_PARAM:",reg_param)
                    print("-------------------------------------------------------------------------")



    print("\n*********************************************************\n")
    print("Final CV Accuracy: ",best_accuracy)
    print("Final Params:    Epochs:",best_params[0],"  LR:",best_params[1],"
REG_PARAM:",best_params[2])

    #----- TESTING ----------------------------

    print("Now we test the model using the best parameter set.")

    X_train = D_train[:, :-1]
    Y_train = D_train[:, -1]


    svm_model = SVM.svm(reg_param=best_params[2])


    #training the SVM
    svm_model.fit(X_train,Y_train,num_epochs=best_params[0],
            learning_rate=best_params[1], show_graph=True, X_val = X_test, Y_val = Y_test)

    #testing
```

```
    _, Y_predicted = svm_model.score( X_test, Y_test)

    print("---- SVM Test Results ----")
    utilities.get_metrics(Y_test, Y_predicted, print_metrics=True)



if __name__ == '__main__':

  print("---Without adding PCA---")
  grid_search_svm(add_PCA = False)
  print("---With PCA---")
  grid_search_svm(add_PCA = True)
```

- **utilities.py**

```
"""
Authors: Berfin Kavşut -  21602459
        Mert Ertuğrul - 21703957
"""
import math
import numpy as np
from sklearn.metrics import precision_score, \
    recall_score, confusion_matrix, classification_report, \
    accuracy_score, f1_score
import matplotlib.pyplot as plt



class Standardizer:

  def __init__(self):
    self.mean_vector = None
    self.std_vector = None

  #function for normalizing with training mean and std
  def standardize_data(self,X, testing=False):

    """
    Standardizes the data by subtracting its mean and dividing by its stdandard deviation

    X: data matrix with each row = one data point
    testing: whether the data is the testing set, in which case the old mean and std values from
    the training set are used

    Features:  to normalize?

    0. age_____yes
    1. sex_____no
    2. cp_____yes
    3. trestbps_yes
```

```
    4.  chol_____yes
    5.  fbs______no
    6.  restecg__no
    7.  thalach__yes
    8.  exang____no
    9.  oldpeak__yes
    10. slope____yes
    11. ca_______yes
    12. thal_____yes

    Label:

    13.  target___no
    """

    if not testing:
      self.mean_vector = np.mean(X, axis=0)
      self.std_vector = np.std(X, axis=0)

    A = (X - self.mean_vector) / self.std_vector

       #some rows should not be standardized
    A[:,1] = X[:,1]
    A[:,5] = X[:,5]
    A[:,6] = X[:,6]
    A[:,8] = X[:,8]
    if X.shape[1]>13:
       A[:,13] = X[:,13]

    np.take(A,np.random.permutation(A.shape[0]),axis=0,out=A)

    return A



def k_fold_split( D, k ):

  """
  Prepares the data for k-fold cross validation, outputs the prepared data
  D: data matrix including features (X) and labels (Y)
  k: number of folds to divide data into
  """

  np.take(D,np.random.permutation(D.shape[0]),axis=0,out=D)
  fold_size = math.ceil( D.shape[0]/k )
  X_folds = []
  Y_folds = []

  X_folds = [ D[ i*fold_size: min( D.shape[0], (i+1)*fold_size ) , :-1] for i in range(k) ]
  Y_folds = [ D[ i*fold_size: min( D.shape[0], (i+1)*fold_size ) , -1] for i in range(k) ]
```

```
  #for i in range(k):
    #print(Y_folds[i].shape)

  return X_folds, Y_folds



def split_train_test(D, test_fraction = 0.1):

  np.take(D,np.random.permutation(D.shape[0]),axis=0,out=D)

  test_idx = int( D.shape[0]*0.1 )

  D_test = D[:test_idx]
  D_train = D[test_idx:]

  return D_train, D_test

def get_metrics(Y_real, Y_predicted, print_metrics=True):

    Y_real = np.where(Y_real == -1, 0, Y_real)
    Y_predicted = np.where(Y_predicted == -1, 0, Y_predicted)
    Y_predicted = (Y_predicted >= 0.5)


    accuracy = accuracy_score(Y_real, Y_predicted)
    precision = precision_score(Y_real, Y_predicted)
    recall = recall_score(Y_real, Y_predicted)
    f1 = f1_score(Y_real, Y_predicted)

    if print_metrics:
        print("Accuracy: ",accuracy)
        print( classification_report(Y_real, Y_predicted) )

        cm = confusion_matrix(Y_real, Y_predicted)
        fig = plt.figure()
        ax = fig.add_subplot(111)
        cax = ax.matshow(cm, cmap='OrRd')
        plt.title('Confusion Matrix')
        fig.colorbar(cax)
        plt.show()


    return accuracy,precision,recall,f1
```

- **main.py**

```
"""
Authors: Berfin Kavşut -  21602459
         Mert Ertuğrul - 21703957
"""
```

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# import pandas_profiling as pp

pd.set_option('display.max_columns', None)
data_heart = pd.read_csv('./heart.csv')

print('Dataset')
print(data_heart.head(5))
print()

print('Information')
data_heart.info()
print()

print('Description')
print(data_heart.describe())
print()

# print(pp.ProfileReport(data_heart))

# Covariance Matrix
fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(data_heart.corr(), annot=True, linewidths=.5, ax=ax)
plt.show()
```