Unsupervised Learning (STA5077Z)

Assignment 2

Natalie Bianca Alexander

Table of contents

1 Abstract

The curse of dimensionality refers to the challenges introduced when analyzing, processing, visualizing, and storing high dimensional data. Dimension reduction was introduced to transform such data, lying on a high-dimensional surface or manifold, to a low-dimensional representation of the data - which retains the important properties of the original data, ideally close to its intrinsic dimension. There are many dimension reduction methods, including (linear and non-linear) principal component analysis, autoencoders and self-organizing maps, all of which are applied in this assignment. These methods often perform well on some data types, but poorly on others. The goal of this assignment is to apply these dimension reduction methods to different kinds of data to determine which methods and parameters work best on different datasets. The KNN algorithm is used to evaluate performance on each dimension reduction method, tested before and after dimension reduction is applied. As a result, I find that a KNN trained on SOM reduced WDBC data had the smallest difference in accuracy (0.005) relative to the original model. In addition, I find that a KNN trained on autoencoder reduced CG data had the smallest absolute difference in accuracy (0.001) relative to the original model.

2 Introduction

Richard E. Bellman, an American mathematician, described the "curse of dimensionality" as the increase in volume associated with adding extra dimensions to a Euclidean space, Karanam (2021) . Although an increasing number of dimensions can theoretically introduce more information to the data, practically, this may also introduce more noise and redundancy during analysis, in addition to an increase in processing requirements.

We also see that as the number of dimensions increase, the machine learning algorithm requires exponentially more

observations to achieve good performance. However, once the optimal number of dimensions have been reached, the model performance begins to deteriorate, Built In (2022) .

Principal Component Analysis (PCA), Self-Organizing Maps (SOM) and Autoencoders are dimension reduction techniques which will be implemented in this assignment.

PCA is a linear dimension reduction method that aims to reduce the dimensionality of a dataset by preserving as much of the original variance in the dataset as possible, Wikipedia (2023b) . Non-linear (or Kernel) PCA on the other hand, is an extension of linear PCA, which allows separation of non-linear data by using Kernel functions Tanner (2022) .

A SOM is a type of artificial neural network used for clustering and visualizing high-dimensional data, GeeksforGeeks (2020) . The SOM learns a low-dimensional, discretized representation of the input data while preserving the topological properties of it, Wikipedia (2023c) .

Autoencoders are neural networks that learn two functions, and have two important components: (1) an encoder which reconstructs the original data to a compressed representation and (2) a decoder that recreates the original data from its encoded representation, Wikipedia (2023a) .

The goal of this assignment is to determine the best dimension reduction method for different kinds of data. In this analysis, we assess two datasets, namely: (1) a sample of not MNIST images containing the letters C and G, and (2) the Wisconsin Diagnostic Breast Cancer dataset.

I will apply the K-nearest neighbors (KNN) algorithm to the data, once before and after applying the dimension reduction method. The performance of each reduction method is then assessed by calculating the change in KNN accuracy pre- and post-dimension reduction.

3 Data

3.1 Wisconsin Diagnostic Breast Cancer Dataset

The Wisconsin Diagnostic Breast Cancer (WDBC) dataset consists of 569 data points classified as either malignant or benign, UCI.edu (2019) . Data was computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. Each instance contains 30 features describing different characteristics of the cell nuclei present in the image.

3.2 CG dataset

The CG dataset consists of a sample of images from the not MNIST dataset, which has letters C and G, Khodabakhsh (2023) . We shall call it the CG dataset, for lack of a better name.

4 Methods

4.1 Exploratory Data Analysis

4.1.0.1 Data Pre-processing

Both the Wisconsin Diagnostic Breast Cancer (WDBC) and the CG datasets were read into R *version 4.3.1*, using Rstudio *2023.09.1+494*.

- The WDBC dataset was in CSV format and so was read into R using read.csv().

- For the CG dataset, I used readPNG() to read in all images in the folders containing the "C" and "G" data, respectively, where the output of each image is a 28 x 28 pixel array. I then converted the dataset to a wide format, where each row represents an observation (image) and each column represents an independent variable (pixels).

For both datasets, the dimensions of the data were assessed to determine the number of observations (rows) and columns (independent and dependent variables). I then checked the column names of each dataset.

- For the WDBC dataset, I renamed variable names to exclude any special characters such as underscores.
- I generated column names for the CG dataset (i.e., pixel 1, pixel 2, etc.)

At this point it was necessary to look at the head and the tail of each dataset to check for any data irregularities. Subsequently, I checked the datatypes of each column and converted discrete and character variables to factors.

- For the WDBC dataset, the ID variable (discrete numeric) and the dependent variable, diagnosis (character) were converted to factors, where the diagnosis variable had two levels, namely: "B" for Benign and "M" for Malignant.
- A "class" dependent variable was added to the C and G dataframes for the CG dataset. The complete CG dataset was then generated by using rbind() to append the G data below the C data, and adding an an ID variable to correctly identify images later on. Both the ID variable and class dependent variable in the combined CG dataset were converted to factors.

I then checked for missing data so that rows containing NA values could be removed.

4.1.0.2 Summary Statistics

I then computed the minimum, Q1, median, mean, Q3 and maximum value for each numeric variable in the WDBC dataset. The minimum and maximum value were only assessed in the CG dataset, to determine the range of pixel values.

4.1.0.3 Variable Distribution

Histograms for each independent variable were generated to illustrate the frequency distribution of observations. In addition, a bar plot was generated to illustrate the frequency of observations in each class of the dependent variable, namely diagnosis ("B" and "M") for the WDBC dataset and the class of letters, "C" and "G" for the CG dataset.

4.1.0.4 CG Image Visualization

Observation 1 in both the C and G classes of the CG dataset were visualized, in addition to observation 1867 in the C class and observation 1866 in the G class. The pixel values of each image were converted back to a 28 x 28 matrix and the transpose of the reversed matrix obtained. Finally the image() function was used to visualize each image.

4.1.0.5 WDBC Independent Variable Correlation

I computed the pairwise Pearson correlation coefficient between each pair of independent variables in the WDBC dataset, using cor(), after which I plotted the upper triangle of the correlation matrix using corrplot(). I used findCorrelation() in the Caret package to find independent variables which are highly correlated with each other, with a correlation coefficient of 0.9 or greater. In each pair of highly correlated independent variables, one variable was removed - since it introduces redundancy.

4.2 Splitting Data into Training and Test Sets

Both the WDBC and CG datasets were split into 70% training and 30% validation sets, using stratified sampling implemented by Caret's createDataPartition() function. Thereafter, the rows of the training data were shuffled using sample() without replacement, so that the model does not "learn" the order of the classes. I then assessed the training sets for class imbalance. If the training set had class imbalance, the minority class would need to be up-sampled or the majority class down-sampled so that the number of observations in the minority class match that of the majority class. Class imbalance in the test set was not of concern, because it represents the natural distribution of classes in the real-word. Thus, I left the test set "unseen".

- For the WDBC dataset, the minority class "M" was up-sampled in the training set to match the frequency of the majority class, "B".
- For the CG dataset, the majority class "G" was down-sampled in the training set to match the frequency of the minority class, "C".

4.3 Data Standardization and Normalization

The WDBC training features were standardized using Z-score normalization, where the training mean is subtracted from each observation and divided by the training standard deviation, so that the resulting features have a mean = 0 and a standard deviation = 1. This was done using the scale() function with scale=TRUE and center=TRUE. The WDBC test data was scaled based on the training data means and standard deviations, where the resultant test mean 0 and standard deviation 1.

The CG training features were already normalized (pixel value divided by 255), resulting in pixel values ranging from 0 to 1.

4.4 KNN Model Construction

4.4.1 Original, High-Dimensional Data

The K-nearest neighbours (KNN) algorithm was applied to the scaled, original (high-dimensional) training datasets, by implementing the knn() function as a parameter in Caret's train() function. A grid search was performed to determine the optimal value of K (ie., number of neighbours) by applying ten-fold cross-validation (CV). The largest mean CV accuracy was determined to find the optimal value of K. Where ties for K existed, the largest value for K was chosen.

- K values 1 to 100 were used in the grid search for the WDBC training dataset.
- K values 1 to 20 were used in the grid search for the CG dataset. The CG training dataset has much more observations and features compared to the WDBC dataset and so increasing the value of K in the grid search in addition to 10-fold cross-validation became computationally expensive. As a result, fewer values of K were explored.

After finding the optimal value of K, the model was re-trained on the entire, (scaled) training dataset, using the optimal value of K as a parameter in Caret's knn3() function. I then made predictions on the scaled, test dataset, using this KNN model. A confusion matrix could then be constructed containing the test predictions and original test labels. Thereafter, the accuracy could be determined by summing the diagonal of the confusion matrix and dividing by the total number of observations.

4.4.2 Principal Component Analysis

Principal Component Analysis (PCA) was applied to the scaled, training data, using prcomp(). Thereafter, I computed the eigenvalues (i.e., variance) of each principal component by squaring the standard deviation of each component. Next, I computed the proportion of explained variance for each component by dividing each component's eigenvalue by the sum of eigenvalues. Scree plots were generated to assess the optimal number of components at the point where the plots elbow off. Finally, I plotted the cumulative proportion of explained variance vs number of components to assess the number of components that cumulatively explain 95% or a greater proportion of variance in the data.

I then created a for-loop to extract i principal components from the training PCA scores, where i represents the number of components from 2 to 6. The reduced data was not reconstructed to the original, feature space because this would be equivalent to training on the original, high-dimensional dataset.

Since the test data should remain unseen, I did not perform PCA directly on the test data. Instead, I predicted the test PCA scores using the PCA model previously trained on the training data.

The KNN models were then trained on the reduced training scores, following the same process as KNN model construction for the original, high-dimensional dataset. By doing so, I could determine the optimal value of K by choosing K with the maximum mean CV accuracy. I could then rebuild the KNN model with the optimal value of K on the training scores and evaluate test accuracy by making predictions on the reduced test scores.

4.4.3 Kernel Principal Component Analysis

The kpca() function was used in Kernel PCA, with the following input arguments: *(1)* the scaled, training features, *(2)* Kernel function (radial or polynomial) *(3)* sigma = 0.01 for the radial Kernel, *(4)* degree = 1, scale = 1 and offset = 1 for the polynomial Kernel, and *(5)* the number of dimensions (2 to 6). Kernel PCA was performed on each unique combination of Kernel function and number of dimensions. The Kernel PCA training scores were then obtained from the model output. The Kernel PCA test scores were computed by making predictions on the scaled, test feature set, using the Kernel PCA model previously built. The KNN models were then trained on each of the reduced PCA training scores, following the same process as KNN model construction and test prediction for the original, high-dimensional dataset and the linear PCA above.

4.4.4 Self-Organizing Maps

Below I describe how models were built at each iteration of a nested for-loop, where each iteration represents a SOM model with a given dimension and topology. Next, I describe a second for-loop, used to train the KNN model on the reduced SOM data.

The goal of the first nested for-loop was to determine the optimal SOM topology (either hexagonal or rectangular) given a dimension (2x2, 3x3, 4x4, 5x5 or 6x6).

- First, the somgrid() function in the Kohonen package was used to initialize the SOM node set, with i x j dimensions, where i and j take on values between 2 and 6 (inclusive of end points) at each iteration. However, it is important to note that at each iteration, i and j are equal e.g., iteration one: 2 x 2 grid, iteration two: 3 x 3 grid, etc. The topology parameter of the somgrid() function took a value of "hexagonal" or "rectangular".
- The som() function was then used to define the SOM model, taking in the following parameters: *(1)* the somgrid previously defined, *(2)* rlen = 500 (the number of times the complete dataset will be presented to the network), and *(3)* alpha = 0.05, 0.01 (a vector of two numbers representing the start and end learning rates).

- I then used the SOM model at a given iteration to predict the reduced, training feature data. The KNN models could then be trained on the reduced SOM (training) data, and ten-fold cross validation performed to determine the optimal value of K, with the largest mean CV accuracy.
- Two KNN models were produced given a i x j dimension: *(1)* a KNN model which was trained on the reduced SOM data with a hexagonal topology, and *(2)* a KNN model trained on the reduced SOM data with a rectangular topology. The mean CV accuracies were then compared between the two KNN models, and the best SOM topology chosen based on the KNN model with the largest mean CV accuracy.

The goal of the next for-loop was to apply SOM with the optimal topology at a given iteration (i x j dimension) and train the KNN model on the resultant, reduced SOM training data.

- First, the SOM model was retrained using the optimal topology
- The SOM model was then used to predict the reduced training and test data.
- The KNN model was then trained on the reduced SOM (training) data, ten-fold cross validation performed to determine the optimal value of K, and the KNN model retrained on the reduced training data using the optimal value of K.
- Predictions were then made on the reduced test data and the test accuracy computed.

The KNN model construction follows the same process as the KNN model construction for the original, high-dimensional dataset and the linear and non-linear PCA described above. Take note that ten-fold cross validation was used in multiple steps, because I optimized for both the topology and value of K.

4.4.5 Autoencoders

Autoencoders were built using Keras and TensorFlow within a Conda environment managed by Anaconda, all executed in RStudio. In RStudio, simply go to Tools > Global Options > Python > Choose Conda environment.

A simple butterfly structure was used for each autoencoder, comprised of an encoder and a decoder. The encoder included the input layer, a hidden layer and the bottleneck layer, where the hidden layer had:

- Either 15, 10 or 5 nodes for the WDBC dataset.
- Either 500, 300 or 100 nodes for the CG dataset.

The decoder included a hidden layer and the output layer, where the decoder's hidden layer had the same number of nodes as the hidden layer in the encoder, above.

In addition, the following hyperparameters were set: *(1)* the input- and output layer had a certain number of nodes equivalent to the number of features in the dataset, *(2)* the bottleneck layer had a certain number of nodes equivalent to the number of dimensions at that iteration, either 2, 3, 4, 5 or 6 and *(3)* the output layer used a Sigmoid activation function. A grid search was performed to determine the optimal number of hidden nodes mentioned above and the optimal hidden layer activation function (either ReLU or Tanh) for each reduced dimension. For each combination of hyperparameters, ten-fold cross-validation was applied when training the autoencoder on the scaled, training set.

The autoencoder was compiled using the mean-squared error objective function and the Adam optimizer. The model could then be fit, using the scaled, training feature data as both the x and y arguments, where 25 epochs were used and a batch size = 128.

I could then evaluate model performance on the validation set and compute the mean CV loss for each model. The optimal hyperparameters for each reduced dimension was determined by the minimum mean CV loss. The autoencoder could then be retrained, as above, but now using the optimal hyperparameters as input. The reduced data was then computed by making predictions on both the training and test sets, using the optimal autoencoder model for a given dimension.

Finally, the KNN model was built as mentioned before for the high dimensional data and all other dimension reduction techniques.

4.5 Model Comparison

For each dataset, the absolute difference in accuracy achieved by the KNN model trained on the original, high dimensional dataset and the each respective dimension reduction technique was computed.

5 Results

5.1 Wisconsin Diagnostic Breast Cancer Dataset

5.1.1 Exploratory Data Analysis

5.1.1.1 Data Pre-processing

There were no missing values in the WDBC dataset, and so the final dataset had 569 rows and 32 columns, where 30 columns represent the independent variables and the remaining 2 columns each represent the ID and the dependent variable (diagnosis). The datatypes of each independent variable and the dependent variable (diagnosis) can be seen in Table 3 in **Appendix A**. We see that all independent variables are continuous and numeric, while the ID variable is discrete and numeric. The dependent variable (diagnosis) is categorical, with two classes namely: "B" for benign and "M" for malignant, which was converted to a factor variable with levels B (0) and M (1). The head of the dataset can be seen in Table 4 to Table 7, while the tail can be seen in Table 8 to Table 11 in **Appendix A**.

5.1.1.2 Summary Statistics

The summary statistics of the WDBC dataset can be seen below in Table 1

We see that the independent variables differ in terms of their range, where certain variables have a narrow range such as "fractal dimension mean" with a range = 0.047 (max = 0.097 and min = 0.050). In contrast, some variables have a wider range such as "area worst" with a range = 4068.800 (max = 4254.000 and min = 185.200). The differences in range suggest variation in the dispersion of the observations for each independent variable. We will need to scale the data later on.

We also see differences in the interquartile range (IQR = Q3 - Q1), where certain variables have a large IQR indicative of central observations that are spread out (e.g., area worst, IQR = 568.700), whereas other variables have a small IQR indicative of central observations that are tightly packed and close to each other (e.g., fractal dimension se, IQR = 0.003).

Most of the independent variables have a right-skewed distribution where the mean is greater than the median. Thus, all independent variables that do not have a symmetric distribution, will have a right-skewed distribution. The following independent variables have a symmetric distribution, where the mean is equal to or approximately equal to

the median: smoothness mean, fractal dimension mean, smoothness se, concave points se, fractal dimension se, and smoothness worst.

Table 1: Summary statistics of the WDBC dataset

Numeric independent variables	Min	Q1	Median	Mean	Q3	Max	Range	IQR
radius mean	6.981	11.700	13.370	14.127	15.780	28.110	21.129	4.080
texture mean	9.710	16.170	18.840	19.290	21.800	39.280	29.570	5.630
perimeter mean	43.790	75.170	86.240	91.969	104.100	188.500	144.710	28.930
area mean	143.500	420.300	551.100	654.889	782.700	2501.000	2357.500	362.400
smoothness mean	0.053	0.086	0.096	0.096	0.105	0.163	0.110	0.019
compactness mean	0.019	0.065	0.093	0.104	0.130	0.345	0.326	0.065
concavity mean	0.000	0.030	0.062	0.089	0.131	0.427	0.427	0.101
concave points mean	0.000	0.020	0.034	0.049	0.074	0.201	0.201	0.054
symmetry mean	0.106	0.162	0.179	0.181	0.196	0.304	0.198	0.034
fractal dimension mean	0.050	0.058	0.062	0.063	0.066	0.097	0.047	0.008
radius se	0.112	0.232	0.324	0.405	0.479	2.873	2.761	0.247
texture se	0.360	0.834	1.108	1.217	1.474	4.885	4.525	0.640
perimeter se	0.757	1.606	2.287	2.866	3.357	21.980	21.223	1.751
area se	6.802	17.850	24.530	40.337	45.190	542.200	535.398	27.340
smoothness se	0.002	0.005	0.006	0.007	0.008	0.031	0.029	0.003
compactness se	0.002	0.013	0.020	0.025	0.032	0.135	0.133	0.019
concavity se	0.000	0.015	0.026	0.032	0.042	0.396	0.396	0.027
concave points se	0.000	0.008	0.011	0.012	0.015	0.053	0.053	0.007
symmetry se	0.008	0.015	0.019	0.021	0.023	0.079	0.071	0.008
fractal dimension se	0.001	0.002	0.003	0.004	0.005	0.030	0.029	0.003
radius worst	7.930	13.010	14.970	16.269	18.790	36.040	28.110	5.780
texture worst	12.020	21.080	25.410	25.677	29.720	49.540	37.520	8.640
perimeter worst	50.410	84.110	97.660	107.261	125.400	251.200	200.790	41.290
area worst	185.200	515.300	686.500	880.583	1084.000	4254.000	4068.800	568.700
smoothness worst	0.071	0.117	0.131	0.132	0.146	0.223	0.152	0.029
compactness worst	0.027	0.147	0.212	0.254	0.339	1.058	1.031	0.192
concavity worst	0.000	0.114	0.227	0.272	0.383	1.252	1.252	0.269
concave points worst	0.000	0.065	0.100	0.115	0.161	0.291	0.291	0.096
symmetry worst	0.156	0.250	0.282	0.290	0.318	0.664	0.508	0.068
fractal dimension worst	0.055	0.071	0.080	0.084	0.092	0.208	0.153	0.021

5.1.1.3 Variable Distribution

Figure 11 to Figure 13 in **Appendix A** shows the histograms illustrating the frequency distribution of observations for each independent variable. These histograms confirm the distributions discussed above in the *Summary Statistics* section.

Figure 14 in **Appendix A** shows the bar plot displaying the frequency of observations in each class of the dependent variable (diagnosis, with "B" and "M" classes). We see class imbalance, where the number of observations in the "B"

class (n = 357) is greater than the number of observations in the "M" class (n = 212). I will need to up-sample my minority class or down-sample my majority class later on so that the classes are balanced in the training set.

5.1.1.4 Independent Variable Correlation

The following independent variables had a Pearson correlation coefficient of 0.9 or greater (shown below in Figure 1). The following highly correlated variables were removed from the dataset:

- **concavity mean** was removed because it is strongly correlated with the variable: *concave points mean.*
- **concave points mean** was removed because it is strongly correlated with variables: *concavity mean* and *concave points worst.*
- **perimeter worst** was removed because it is strongly correlated with variables : *radius mean, perimeter mean, area mean, radius worst*, and *area worst.*
- **radius worst** was removed because it is strongly correlated with variables: *radius mean, perimeter mean, area mean, perimeter worst* and *area worst.*
- **perimeter mean** was removed because it is strongly correlated with variables: *radius mean, area mean, radius worst, perimeter worst* and *area worst.*
- **area worst** was removed because it is strongly correlated with variables: *radius mean, perimeter mean, area mean, radius worst* and *perimeter worst.*
- **radius mean** was removed because it is strongly correlated with variables: *perimeter mean, area mean, radius worst, perimeter worst* and *area worst.*
- **perimeter se** was removed because it is strongly correlated with variables: *radius se,* and *area mean.*
- **area se** was removed because it is strongly correlated with variables *radius se*, and *perimeter se.*
- **texture mean** was removed because it is strongly correlated with the variable: *texture worst.*

As a result, 20 independent variables remained in the WDBC dataset.

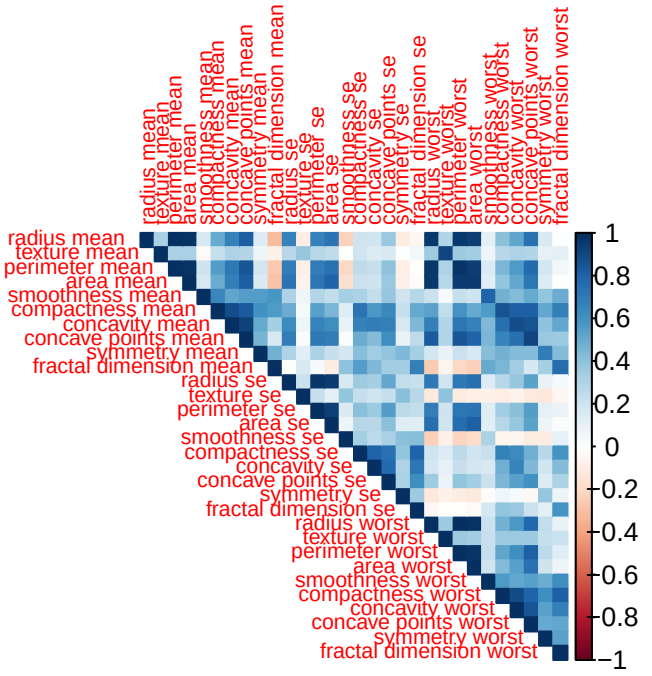

Figure 1: Pearson correlation matrix of independent variables in the WDBC dataset

5.1.2 Splitting Data into Training and Test Sets

After splitting the CG data into 70% training and 30% test sets, I checked for class imbalance. The test set had 107 observations in the "B" class and 63 observations in the "M" class, however, the test set is left "untouched" and "unseen" so that it represents the true real-world distribution . On the other hand, the "B" class in the training dataset had 250 observations, while the "M" class had 149 observations. I decided to up-sample the minority class ("M") by randomly sampling observations from the minority class so that the number of observations in the minority class matches that of the majority class ("B"). Up-sampling was done by using Caret's upSample() function. As a result, the training classes both had 250 observations.

5.1.3 Data Standardization

After standardizing the data, each feature in the training data had a mean = 0 and standard deviation = 1, while each feature in the test data had a approximate mean of 0 and approximate standard deviation of 1, as expected.

5.1.4 KNN Model Using High Dimensional Data

5.1.4.1 Determine the Optimal Value of K

The optimal value of K is 7 with a mean CV accuracy of 0.958 or (95.8%) when the KNN() algorithm is applied to the scaled, original (high-dimensional) WDBC training dataset, implementing 10-fold cross-validation as shown in Figure 2 below.

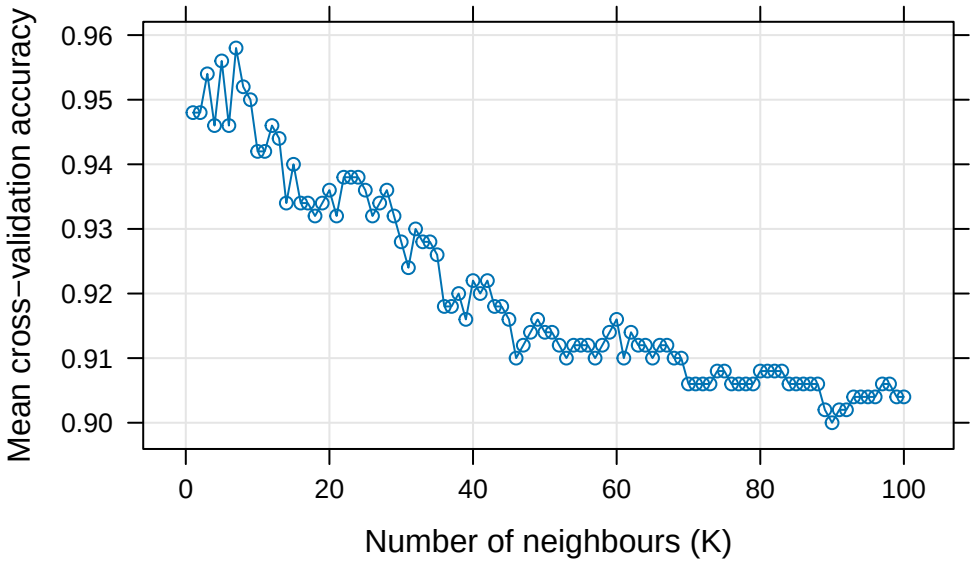

Figure 2: Mean Cross-Validation accuracy for each value of K: 1 to 100. Here 10-fold cross-validation was applied to the WDBC dataset, when training the KNN on the orginal-scaled, high-dimensionsal data

5.1.4.2 Model Performance

The optimal value of K = 7 was then used to rebuild the KNN model on the original, high-dimensional training dataset as discussed in the methods section. The model was then applied to the test set, achieving a 0.929 (92.9%) test accuracy on the original, high dimensional WDBC test data. The confusion matrix can be seen in Table 12 of **Appendix A.** The test accuracy of the original, high-dimensional dataset can be seen in Table 2 under the *Model Comparison* sub-section of the *Results* section.

5.1.5 KNN Model Using Principal Component Analysis

5.1.5.1 Principal Component Analysis Results

Twenty principal components were generated, since I had 20 independent variables, after removing highly correlated variables.

Figure 3 below shows the results of PCA, where screeplots in (a) and (b) elbow off at 5 principal components, suggesting that (a) the optimal eigenvalue is below 2.5 and (b) the optimal proportion of explained variance is below 0.1, respectively. However, the cumulative proportion of explained variance in plot (c) suggests that the optimal number of principal components is 10, where PC = 10 explains at least 95% of variance in the data.

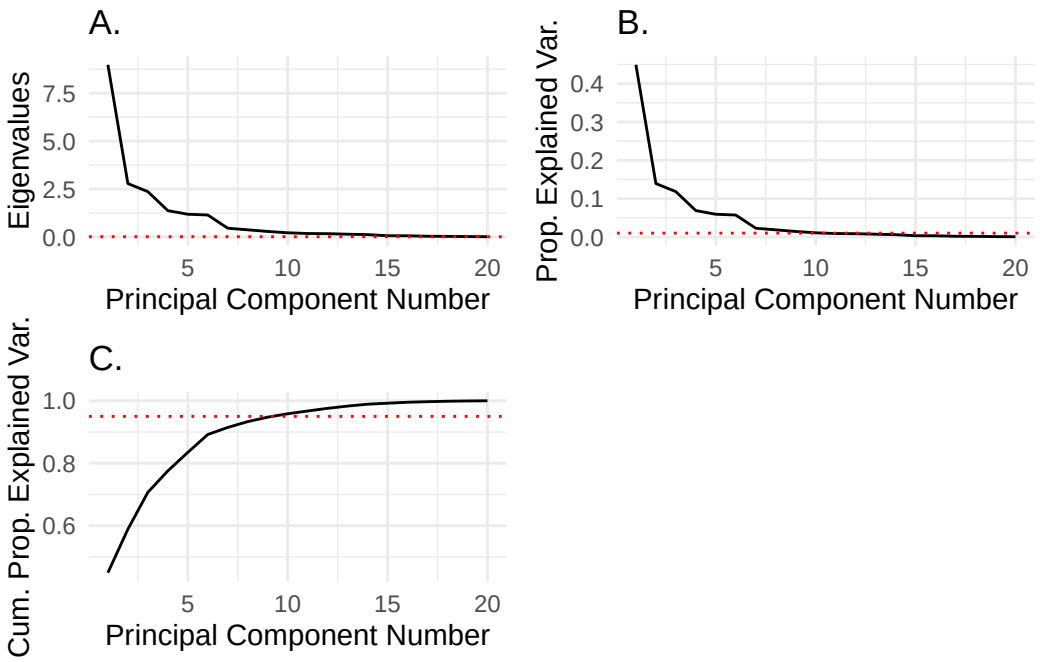

Figure 3: Principal Component Analysis on scaled WDBC training data. (a) Number of principal components vs Eigenvalues, (b) Number of principal components vs Proportion of explained variance and (c) Number of principal components vs Cumulative proportion of explained variance

5.1.5.2 Model Performance

The optimal value of K for each reduced dimension is shown below in Figure 4 . Table 2 shows the results of PCA.

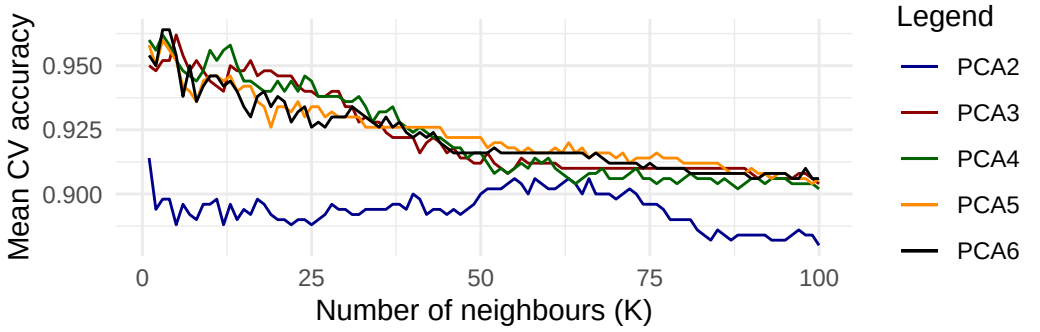

Figure 4: Mean cross-validation accuracy for neighbours k = 1 to 100, performed on the validation WDBC PCA scores.

5.1.6 KNN Model Using Kernel Principal Component Analysis

5.1.6.1 Model Performance

Figure 5 below shows the optimal value of K for each KNN model. Table 2 shows the results of Kernel PCA.

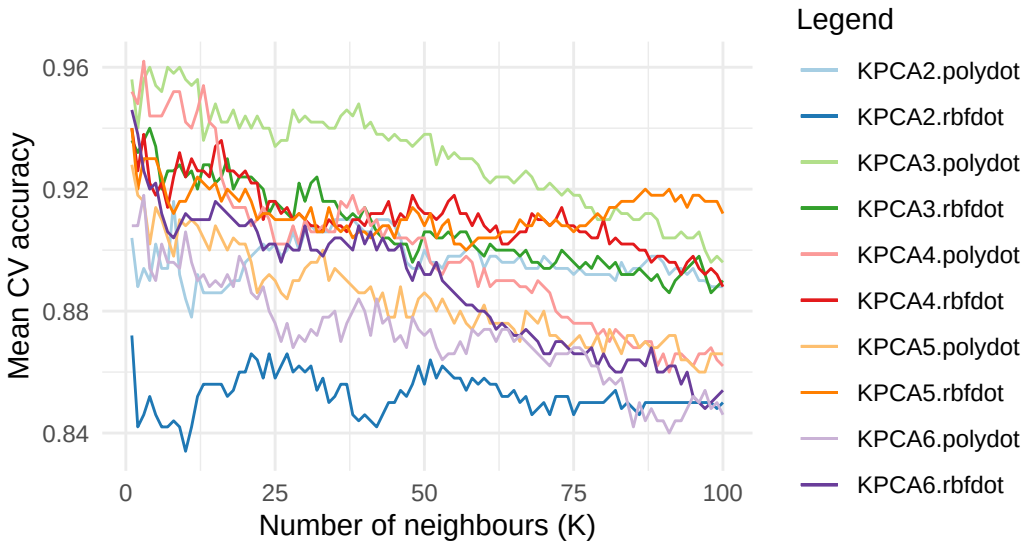

Figure 5: Mean cross-validation accuracy for neighbours k = 1 to 100, performed on the validation WDBC Kernel PCA scores

5.1.7 KNN Model Using Self-Organizing Maps

5.1.7.1 Model Performance

Figure 6 below shows the optimal value of K for each KNN model trained on the reduced SOM data. Table 2 shows the results of the SOM models.

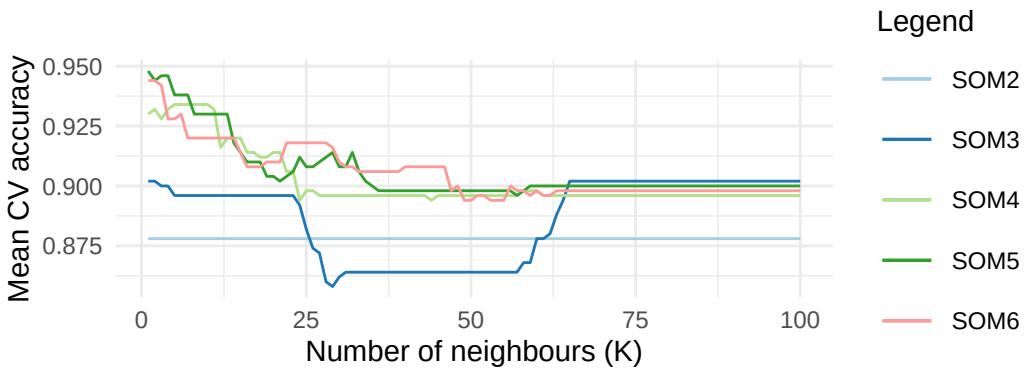

Figure 6: Mean cross-validation accuracy for neighbours k = 1 to 100, performed on the reduced WDBC SOM validation data

5.1.8 KNN Model Using Autoencoders

5.1.8.1 Model Performance

Figure 7 below shows the optimal value of K for each KNN model trained on the reduced autoencoder data. Table 2 shows the results of the autoencoders.

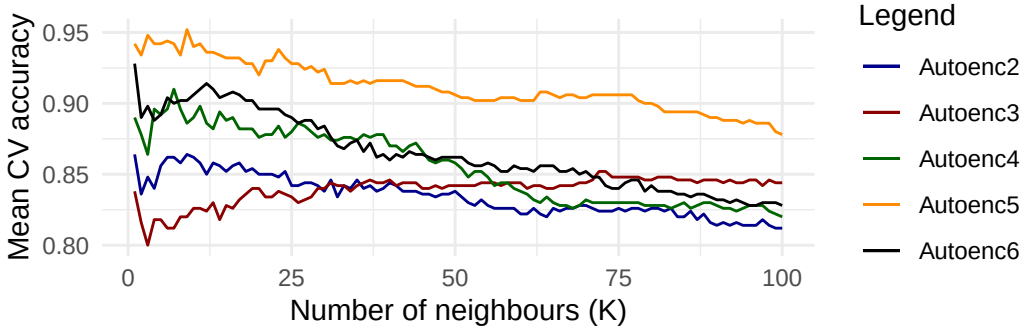

Figure 7: Mean cross-validation accuracy for neighbours k = 1 to 100, performed on the reduced WDBC autoencoder validation data.

5.2 CG dataset

5.2.1 Exploratory Data Analysis

5.2.1.1 Data Pre-processing

There were no missing values in the CG dataset, and so the final processed dataset had 1873 images (rows) x 784 pixels (columns) in the "C" class and 1872 images (rows) and 784 pixels (columns) in the "G" class, where each set of

pixels form a 28 x 28 matrix. The bar plot in Figure 17 of **Appendix B** shows the frequency of each class. The head (Table 13 : C and Table 14 : G) and tail (Table 15 : C and Table 16 : G) of each class can be seen in **Appendix B.**

5.2.1.2 Summary Statistics

We see that the minimum pixel value is 0 and the maximum pixel value is 1, suggesting that the dataset is already standardized. In other words, each pixel value has been divided by 255.

5.2.1.3 Variable Distribution

The histograms in Figure 15 of **Appendix B** shows the frequency distribution of pixel values across all images. We see that there is a relatively high proportion of pixel values = 0 and 1. This suggests that the images are black (0) and white (1), and so we visualize the images using grayscale (2 pixel values).

5.2.1.4 CG Image visualization

Figure 16 in **Appendix B** below shows a sample of images, where the normalized, pixel values are plotted against the x- and y-axis.

5.2.2 Splitting the Data into Training and Test Sets

After splitting the CG data into 70% training and 30% test sets, I checked for class imbalance. The classes in the test set each had 561 observations. However, the "C" class in the training dataset had 1312 observations, while the "G" class had 1311 observations. I decided to down-sample the majority "C" class by randomly selecting and removing an observation (observation ID = c619) which was in the "C" class.

5.2.3 Data Standardization

The pixel values were already normalized to have values between 0 and 1. This means readPNG() already divided pixel values by 255.

5.2.4 KNN Model Using High Dimensional Data

5.2.4.1 Determine the Optimal Value of K

The optimal value of K is 15 with a mean CV accuracy of 0.918 (or 91.8%) when the KNN() algorithm is applied to the scaled (original, high-dimensional) CG training dataset, implementing 10-fold cross-validation as shown in Figure 8 below.

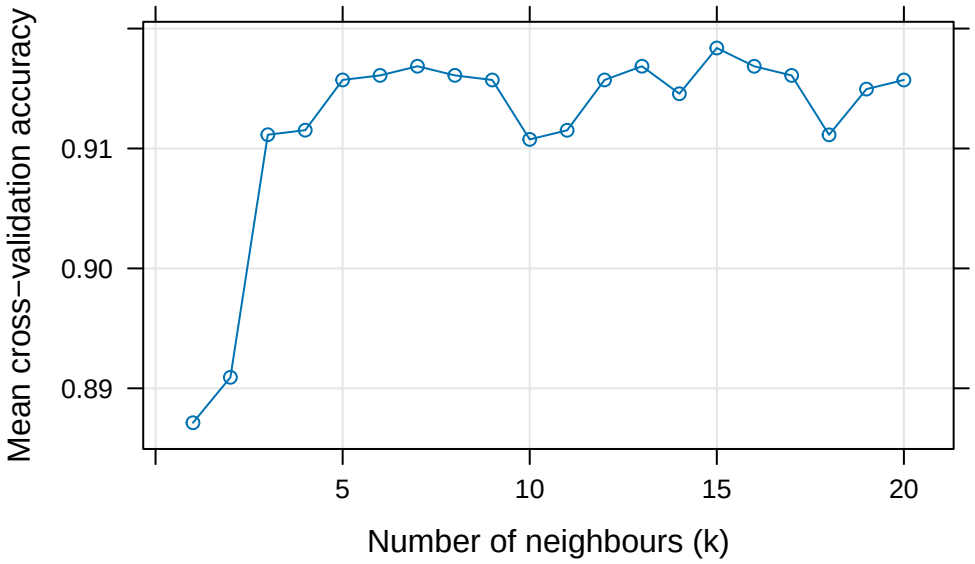

Figure 8: Mean Cross-Validation accuracy for each value of K = 1 to 20. Here 10-fold cross-validation was applied to the original, high-dimensional CG dataset

5.2.4.2 Model Performance

The optimal value of K = 15 was then used to rebuild the KNN model on the original, high-dimensional dataset as discussed in the methods section. The model was then applied to the test set, achieving a 0.933 (93.3%) test accuracy on the original, high dimensional CG test data. The confusion matrix can be seen in Table 17 of **Appendix B.** The test accuracy on the original, high-dimensional dataset can be seen in Table 2 .

5.2.5 KNN Model Using Principal Component Analysis

5.2.5.1 Principal Component Analysis

Seven Hundred and eighty four principal components were generated, since I had 784 independent variables (pixels).

Figure 9 below shows the results of PCA, where screeplots in (a) and (b) elbow off at about 50 components, suggesting that (a) the optimal eigenvalue is less than 5 and (b) the proportion of explained variance is below 0.05, respectively. However, the cumulative proportion of explained variance plot in (c) suggests that the optimal number of principal components is 100, where PC = 100 explains at least 95% of variance in the data.

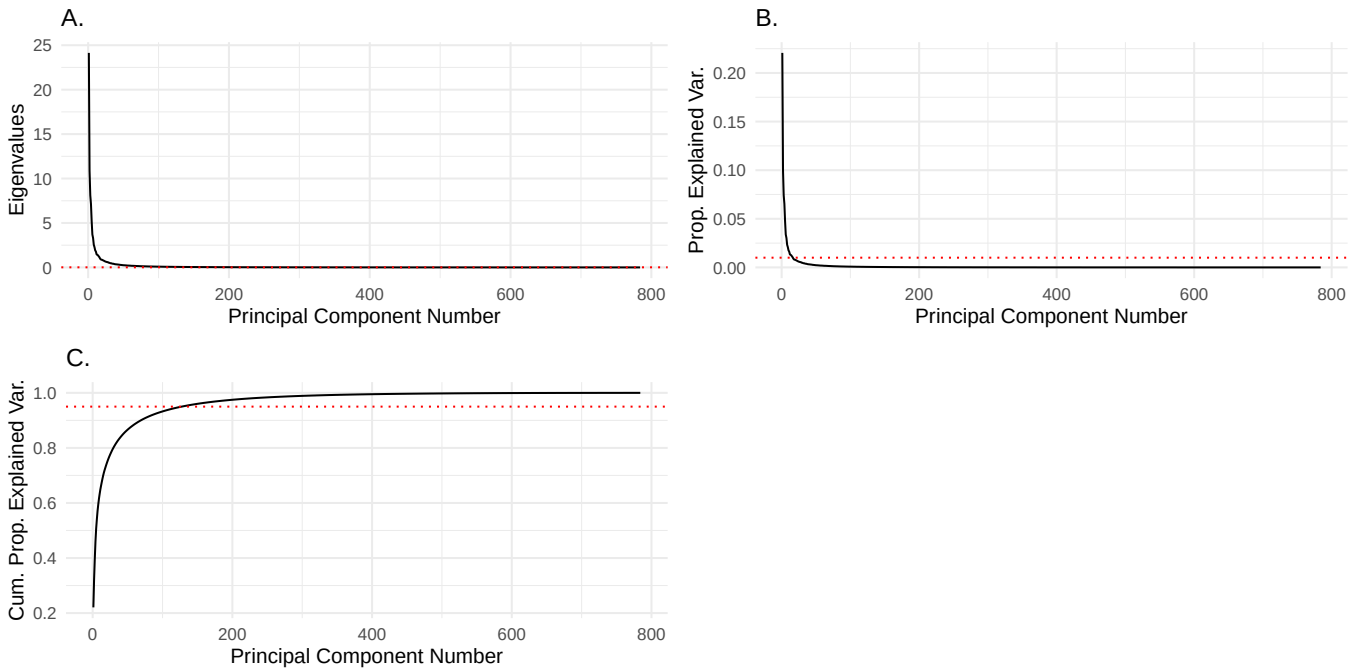

Figure 9: Principal Component Analysis on the scaled CG training data. (a) Number of principal components vs Eigenvalues, (b) Number of principal components vs Proportion of explained variance and (c) Number of principal components vs Cumulative proportion of explained variance.

5.2.5.2 Model Performance

The optimal value of K for each reduced dimension is shown below in Figure 10 . Table 2 shows the results of PCA.

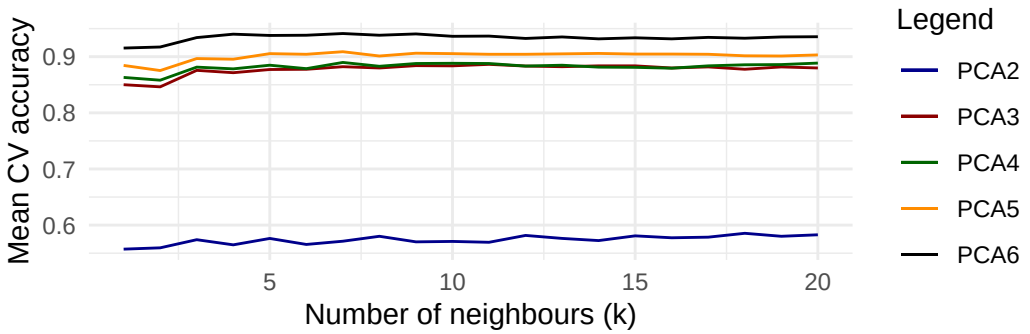

Figure 10: Mean cross-validation accuracy for K = 1 to 20, performed on the CG validation PCA scores

5.2.6 KNN Model Using Kernel Principal Component Analysis, Self-Organizing Maps and Autoencoders

Table 2 shows the results of Kernel PCA, SOM and Autoencoder dimension reduction applied to the CG dataset. The KNN cross-validation plots are not shown for these methods, because similar to Figure 10 for PCA, we do not see much fluctuation in the optimal value of K.

5.3 Model Comparison

5.4 Wisconsin Diagnostic Breast Cancer Dataset

In Table 2, we see that the test accuracy of the original, high dimensional dataset (model *wdbc.original.K7* with K = 7) is 0.929, which is also the largest test accuracy among the WDBC results. The following (optimal) dimension reduction methods had a similar model performance to the original (optimal) model (ranked from most similar to least similar): SOM, Kernel PCA, autoencoders and linear PCA.

For the PCA results, we see that model *wdbc.pca.K5.D3* (with K = 5 and dimensions = 3) had the smallest absolute difference in accuracy (0.205) relative to the original model. For the Kernel PCA models, we that the model *wdbc.kpca.K1.rbf.D6* (with K = 1, a radial Kernel and dimensions = 6) had the smallest absolute difference in accuracy (0.035) relative to the original model. For the autoencoders, we see that model *wdbc.autoen.K9.D5* (with K = 9, dimensions = 5 nodes in the bottleneck layer, activation function = Tanh and number of nodes in the hidden layers = 15) had the smallest absolute difference in accuracy (0.041) relative to the original model. For the SOM models, we see that model *wdbc.som.K2.rec.D6* (with K = 2, dimensions = 6x6, and topology = rectangular) had the smallest absolute difference in accuracy (0.005) relative to the original model.

5.5 CG Dataset

In Table 2, we see that the test accuracy of the original, high dimensional dataset (model *cg.original.K15* with K = 15) is 0.933, which is also the third largest test accuracy among the CG results. Model *cg.pca.K7.D6* which uses PCA, with K = 7 and dimensions = 6 had the largest test accuracy of 0.955, followed by model *cg.autoen.K7.D6*, which uses an autoencoder, with K =7 and dimensions = 6 (test accuracy = 0.934). The following (optimal) dimension reduction methods had a similar model performance to the original (optimal) model (from most similar to least similar): autoencoders, Kernel PCA, PCA and the SOM.

For the PCA results, we see that model *cg.pca.K7.D5* (with K = 7 and dimensions = 5) had the smallest absolute difference in accuracy (0.016) relative to the original model. For the Kernel PCA models, we that the model *cg.kpca.K7.poly.D6* (with K = 7, a polynomial Kernel and dimensions = 6) had the smallest absolute difference in accuracy (0.013) relative to the original model. For the autoencoders, we see that model *cg.autoen.K7.D6* (with K = 7, dimensions = 6 nodes in the bottleneck layer, activation function = ReLU and number of nodes in the hidden layers = 300) had the smallest absolute difference in accuracy (0.001) relative to the original model. For the SOM models, we see that model *cg.som.K20.hex.D6* (with K = 20, dimensions = 6x6, and topology = hexagonal) had the smallest absolute difference in accuracy (0.075) relative to the original model.

Table 2: Model Performance and Comparison of KNN Trained on Reduced WDBC and CG Data Obtained by Various Dimension Reduction Methods.

Model ID	Model Parameters	Test Accuracy	Difference
wdbc.original.K7	K = 7	0.929	0.000
wdbc.pca.K1.D2	K = 1	0.712	0.217
wdbc.pca.K5.D3	K = 5	0.724	0.205
wdbc.pca.K3.D4	K = 3	0.700	0.229
wdbc.pca.K3.D5	K = 3	0.700	0.229
wdbc.pca.K4.D6	K = 4	0.529	0.400
wdbc.kpca.K1.rbf.D2	K = 1, Kernel = rbfdot, sigma = 0.1, offset = NA	0.729	0.200
wdbc.kpca.K4.rbf.D3	K = 4, Kernel = rbfdot, sigma = 0.1, offset = NA	0.800	0.129
wdbc.kpca.K1.rbf.D4	K = 1, Kernel = rbfdot, sigma = 0.1, offset = NA	0.829	0.100
wdbc.kpca.K1.rbf.D5	K = 1, Kernel = rbfdot, sigma = 0.1, offset = NA	0.853	0.076
wdbc.kpca.K1.rbf.D6	K = 1, Kernel = rbfdot, sigma = 0.1, offset = NA	0.894	0.035
wdbc.kpca.K8.poly.D2	K = 8, Kernel = polydot, degree = 1, offset = 1, scale = 1	0.812	0.117
wdbc.kpca.K9.poly.D3	K = 9, Kernel = polydot, degree = 1, offset = 1, scale = 1	0.841	0.088
wdbc.kpca.K3.poly.D4	K = 3, Kernel = polydot, degree = 1, offset = 1, scale = 1	0.847	0.082
wdbc.kpca.K1.poly.D5	K = 1, Kernel = polydot, degree = 1, offset = 1, scale = 1	0.847	0.082
wdbc.kpca.K3.poly.D6	K = 3, Kernel = polydot, degree = 1, offset = 1, scale = 1	0.771	0.158
wdbc.autoen.K9.D2	K = 9, Activation function = tanh. No. of nodes in hidden layers = 15	0.788	0.141
wdbc.autoen.K73.D3	K = 73, Activation function = relu. No. of nodes in hidden layers = 15	0.782	0.147
wdbc.autoen.K7.D4	K = 7, Activation function = tanh. No. of nodes in hidden layers = 15	0.835	0.094
wdbc.autoen.K9.D5	K = 9, Activation function = tanh. No. of nodes in hidden layers = 15	0.888	0.041
wdbc.autoen.K1.D6	K = 1, Activation function = tanh. No. of nodes in hidden layers = 15	0.871	0.058
wdbc.som.K100.hex.D2	K = 100, topology = hexagonal, rlen = 500, alpha = [0.05, 0.01]	0.882	0.047
wdbc.som.K100.rec.D3	K = 100, topology = rectangular, rlen = 500, alpha = [0.05, 0.01]	0.906	0.023
wdbc.som.K10.rec.D4	K = 10, topology = rectangular, rlen = 500, alpha = [0.05, 0.01]	0.847	0.082
wdbc.som.K1.hex.D5	K = 1, topology = hexagonal, rlen = 500, alpha = [0.05, 0.01]	0.918	0.011
wdbc.som.K2.rec.D6	K = 2, topology = rectangular, rlen = 500, alpha = [0.05, 0.01]	0.924	0.005
cg.original.K15	K = 15	0.933	0.000
cg.pca.K18.D2	K = 18	0.596	0.337
cg.pca.K11.D3	K = 11	0.894	0.039
cg.pca.K7.D4	K = 7	0.889	0.044
cg.pca.K7.D5	K = 7	0.917	0.016
cg.pca.K7.D6	K = 7	0.955	0.022
cg.kpca.K12.rbf.D2	K = 12, Kernel = rbfdot, sigma = 0.1, offset = NA	0.498	0.435
cg.kpca.K17.rbf.D3	K = 17, Kernel = rbfdot, sigma = 0.1, offset = NA	0.573	0.360
cg.kpca.K6.rbf.D4	K = 6, Kernel = rbfdot, sigma = 0.1, offset = NA	0.545	0.388
cg.kpca.K19.rbf.D5	K = 19, Kernel = rbfdot, sigma = 0.1, offset = NA	0.635	0.298

Model ID	Model Parameters	Test Accuracy	Difference
cg.kpca.K16.rbf.D6	K = 16, Kernel = rbfdot, sigma = 0.1, offset = NA	0.644	0.289
cg.kpca.K9.poly.D2	K = 9, Kernel = polydot, degree = 1, offset = 1, scale = 1	0.534	0.399
cg.kpca.K18.poly.D3	K = 18, Kernel = polydot, degree = 1, offset = 1, scale = 1	0.874	0.059
cg.kpca.K16.poly.D4	K = 16, Kernel = polydot, degree = 1, offset = 1, scale = 1	0.870	0.063
cg.kpca.K9.poly.D5	K = 9, Kernel = polydot, degree = 1, offset = 1, scale = 1	0.881	0.052
cg.kpca.K7.poly.D6	K = 7, Kernel = polydot, degree = 1, offset = 1, scale = 1	0.920	0.013
cg.autoen.K6.D2	K = 6, Activation function = relu. No. of nodes in hidden layers = 500	0.519	0.414
cg.autoen.K12.D3	K = 12, Activation function = relu. No. of nodes in hidden layers = 300	0.637	0.296
cg.autoen.K9.D4	K = 9, Activation function = relu. No. of nodes in hidden layers = 500	0.801	0.132
cg.autoen.K14.D5	K = 14, Activation function = relu. No. of nodes in hidden layers = 300	0.948	0.015
cg.autoen.K7.D6	K = 7, Activation function = relu. No. of nodes in hidden layers = 300	0.934	0.001
cg.som.K20.hex.D2	K = 20, topology = hexagonal, rlen = 500, alpha = [0.05, 0.01]	0.504	0.429
cg.som.K14.hex.D3	K = 14, topology = hexagonal, rlen = 500, alpha = [0.05, 0.01]	0.754	0.179
cg.som.K20.hex.D4	K = 20, topology = hexagonal, rlen = 500, alpha = [0.05, 0.01]	0.793	0.140
cg.som.K20.hex.D5	K = 20, topology = hexagonal, rlen = 500, alpha = [0.05, 0.01]	0.850	0.083
cg.som.K20.hex.D6	K = 20, topology = hexagonal, rlen = 500, alpha = [0.05, 0.01]	0.858	0.075

Note: ^^ Model ID:

Note: [1] Data source: cg or wdbc [2] Data type: original, pca, kpca, autoen or som [3] No. neighbours: K# [4] Kernel: rbf or poly [5] Topology: hex or rec [6] Dimension: D#

6 Discussion

We conclude that the SOM dimension reduction method is the most suitable for the WDBC dataset (a smaller n x p dataset) when assessing the KNN model performance. In addition, we conclude that the autoencoder dimension reduction method is the most suitable for the CG dataset (a larger n x p dataset) when assessing the KNN model performance.

We also take note that the best performing KNN model trained on the WDBC reduced data, was also the best performing SOM model (with dimensions = 6 x 6), suggesting that larger dimensions are preferred during KNN model training on reduced SOM data. We see a similar trend for the best performing KNN models trained on other forms of WDBC reduced data, where the dimensions are 3 or greater. Furthermore, we see that the optimal SOM model required a smaller number of neighbours (K = 2) compared to the KNN model trained on the original data which required a larger number of neighbours (K = 7) to achieve a similar level of accuracy. All other optimal KNN models trained on the reduced WDBC data had a value of K less than the value of K of the original data, with the exception of the autoencoder model with K = 9.

The best performing KNN model on the CG reduced data, was also the best performing autoencoder model (with

dimensions = 6 nodes in the bottleneck layer), also suggesting that larger dimensions are preferred during KNN model training on reduced autoencoder data. We see a similar trend for the best performing KNN models trained on the other forms of reduced CG data, where the dimensions are either 5 or 6. Furthermore, we see that the optimal autoencoder model required a smaller value of neighbours (K = 7) compared to the KNN model trained on the original data which required a larger K (K = 15) in order to achieve a similar accuracy. All other optimal KNN models trained on reduced CG data had a value of K less than the value of K of the original data, with the exception of the SOM model with K = 20.

These results suggest that KNN models should train on data that has the optimal dimensionality to capture both the low-dimensional structure (small D) and the intrinsic qualities of the data (larger D), regardless of the number of features or observations. In addition, it is evident that high-dimensional data requires large values of K for KNN models, however, small values of K are preferable. Small values of K can be achieved by reducing the data to its low-dimensional representation.

In future, I would like to look at how these models perform when training on data that has multinomial classes. I would then need to assess other metrics, besides accuracy to avoid a model which is biased towards the prediction of certain classes.

7 Conclusion

In this assignment, I explored the use of various dimension reduction methods such as linear and non-linear PCA, autoencoders and SOM on different kinds of data. Based on my results, I recommend using the autoencoder dimension reduction method to classify the two letters in the CG dataset, using the KNN algorithm. I also recommend using the SOM dimension reduction method to classify the breast cell nuclei in the WDBC dataset as either malignant or benign, using the KNN algorithm.

8 Appendix A

8.1 WDBC Dataset

Table 3: WDBC variable- and data- types

Variable name	Variable type	Data type
id	ID	Categorical
diagnosis	Dependent	Categrical
radius mean	Independent	Continuous
texture mean	Independent	Continuous
perimeter mean	Independent	Continuous
area mean	Independent	Continuous
smoothness mean	Independent	Continuous
compactness mean	Independent	Continuous
concavity mean	Independent	Continuous
concave points mean	Independent	Continuous
symmetry mean	Independent	Continuous
fractal dimension mean	Independent	Continuous
radius se	Independent	Continuous
texture se	Independent	Continuous
perimeter se	Independent	Continuous
area se	Independent	Continuous
smoothness se	Independent	Continuous
compactness se	Independent	Continuous
concavity se	Independent	Continuous
concave points se	Independent	Continuous
symmetry se	Independent	Continuous
fractal dimension se	Independent	Continuous
radius worst	Independent	Continuous
texture worst	Independent	Continuous
perimeter worst	Independent	Continuous
area worst	Independent	Continuous
smoothness worst	Independent	Continuous
compactness worst	Independent	Continuous
concavity worst	Independent	Continuous
concave points worst	Independent	Continuous
symmetry worst	Independent	Continuous
fractal dimension worst	Independent	Continuous

Table 4: Head of WDBC dataset, with columns 1 to 8

id	diagnosis	radius mean	texture mean	perimeter mean	area mean	smoothness mean	compactness mean
842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760
842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864
84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990
84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390
84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280
843786	M	12.45	15.70	82.57	477.1	0.12780	0.17000

Table 5: Head of WDBC dataset, with columns 9 to 16

concavity mean	concave points mean	symmetry mean	fractal dimension mean	radius se	texture se	perimeter se	area se
0.3001	0.14710	0.2419	0.07871	1.0950	0.9053	8.589	153.40
0.0869	0.07017	0.1812	0.05667	0.5435	0.7339	3.398	74.08
0.1974	0.12790	0.2069	0.05999	0.7456	0.7869	4.585	94.03
0.2414	0.10520	0.2597	0.09744	0.4956	1.1560	3.445	27.23
0.1980	0.10430	0.1809	0.05883	0.7572	0.7813	5.438	94.44
0.1578	0.08089	0.2087	0.07613	0.3345	0.8902	2.217	27.19

Table 6: Head of WDBC dataset, with columns 17 to 24

smoothness se	compactness se	concavity se	concave points se	symmetry se	fractal dimension se	radius worst	texture worst
0.006399	0.04904	0.05373	0.01587	0.03003	0.006193	25.38	17.33
0.005225	0.01308	0.01860	0.01340	0.01389	0.003532	24.99	23.41
0.006150	0.04006	0.03832	0.02058	0.02250	0.004571	23.57	25.53
0.009110	0.07458	0.05661	0.01867	0.05963	0.009208	14.91	26.50
0.011490	0.02461	0.05688	0.01885	0.01756	0.005115	22.54	16.67
0.007510	0.03345	0.03672	0.01137	0.02165	0.005082	15.47	23.75

Table 7: Head of WDBC dataset, with columns 25 to 32

perimeter worst	area worst	smoothness worst	compactness worst	concavity worst	concave points worst	symmetry worst	fractal dimension worst
184.60	2019.0	0.1622	0.6656	0.7119	0.2654	0.4601	0.11890
158.80	1956.0	0.1238	0.1866	0.2416	0.1860	0.2750	0.08902
152.50	1709.0	0.1444	0.4245	0.4504	0.2430	0.3613	0.08758
98.87	567.7	0.2098	0.8663	0.6869	0.2575	0.6638	0.17300

perimeter worst	area worst	smoothness worst	compactness worst	concavity worst	concave points worst	symmetry worst	fractal dimension worst
152.20	1575.0	0.1374	0.2050	0.4000	0.1625	0.2364	0.07678
103.40	741.6	0.1791	0.5249	0.5355	0.1741	0.3985	0.12440

Table 8: Tail of WDBC dataset, with columns 1 to 8

	id	diagnosis	radius mean	texture mean	perimeter mean	area mean	smoothness mean	compactness mean
564	926125	M	20.92	25.09	143.00	1347.0	0.10990	0.22360
565	926424	M	21.56	22.39	142.00	1479.0	0.11100	0.11590
566	926682	M	20.13	28.25	131.20	1261.0	0.09780	0.10340
567	926954	M	16.60	28.08	108.30	858.1	0.08455	0.10230
568	927241	M	20.60	29.33	140.10	1265.0	0.11780	0.27700
569	92751	B	7.76	24.54	47.92	181.0	0.05263	0.04362

Table 9: Tail of WDBC dataset, with columns 9 to 16

	concavity mean	concave points mean	symmetry mean	fractal dimension mean	radius se	texture se	perimeter se	area se
564	0.31740	0.14740	0.2149	0.06879	0.9622	1.026	8.758	118.80
565	0.24390	0.13890	0.1726	0.05623	1.1760	1.256	7.673	158.70
566	0.14400	0.09791	0.1752	0.05533	0.7655	2.463	5.203	99.04
567	0.09251	0.05302	0.1590	0.05648	0.4564	1.075	3.425	48.55
568	0.35140	0.15200	0.2397	0.07016	0.7260	1.595	5.772	86.22
569	0.00000	0.00000	0.1587	0.05884	0.3857	1.428	2.548	19.15

Table 10: Tail of WDBC dataset, with columns 17 to 24

	smoothness se	compactness se	concavity se	concave points se	symmetry se	fractal dimension se	radius worst	texture worst
564	0.006399	0.04310	0.07845	0.02624	0.02057	0.006213	24.290	29.41
565	0.010300	0.02891	0.05198	0.02454	0.01114	0.004239	25.450	26.40
566	0.005769	0.02423	0.03950	0.01678	0.01898	0.002498	23.690	38.25
567	0.005903	0.03731	0.04730	0.01557	0.01318	0.003892	18.980	34.12
568	0.006522	0.06158	0.07117	0.01664	0.02324	0.006185	25.740	39.42
569	0.007189	0.00466	0.00000	0.00000	0.02676	0.002783	9.456	30.37

Table 11: Tail of WDBC dataset, with columns 25 to 32

	perimeter worst	area worst	smoothness worst	compactness worst	concavity worst	concave points worst	symmetry worst	fractal dimension worst
564	179.10	1819.0	0.14070	0.41860	0.6599	0.2542	0.2929	0.09873
565	166.10	2027.0	0.14100	0.21130	0.4107	0.2216	0.2060	0.07115
566	155.00	1731.0	0.11660	0.19220	0.3215	0.1628	0.2572	0.06637
567	126.70	1124.0	0.11390	0.30940	0.3403	0.1418	0.2218	0.07820
568	184.60	1821.0	0.16500	0.86810	0.9387	0.2650	0.4087	0.12400
569	59.16	268.6	0.08996	0.06444	0.0000	0.0000	0.2871	0.07039

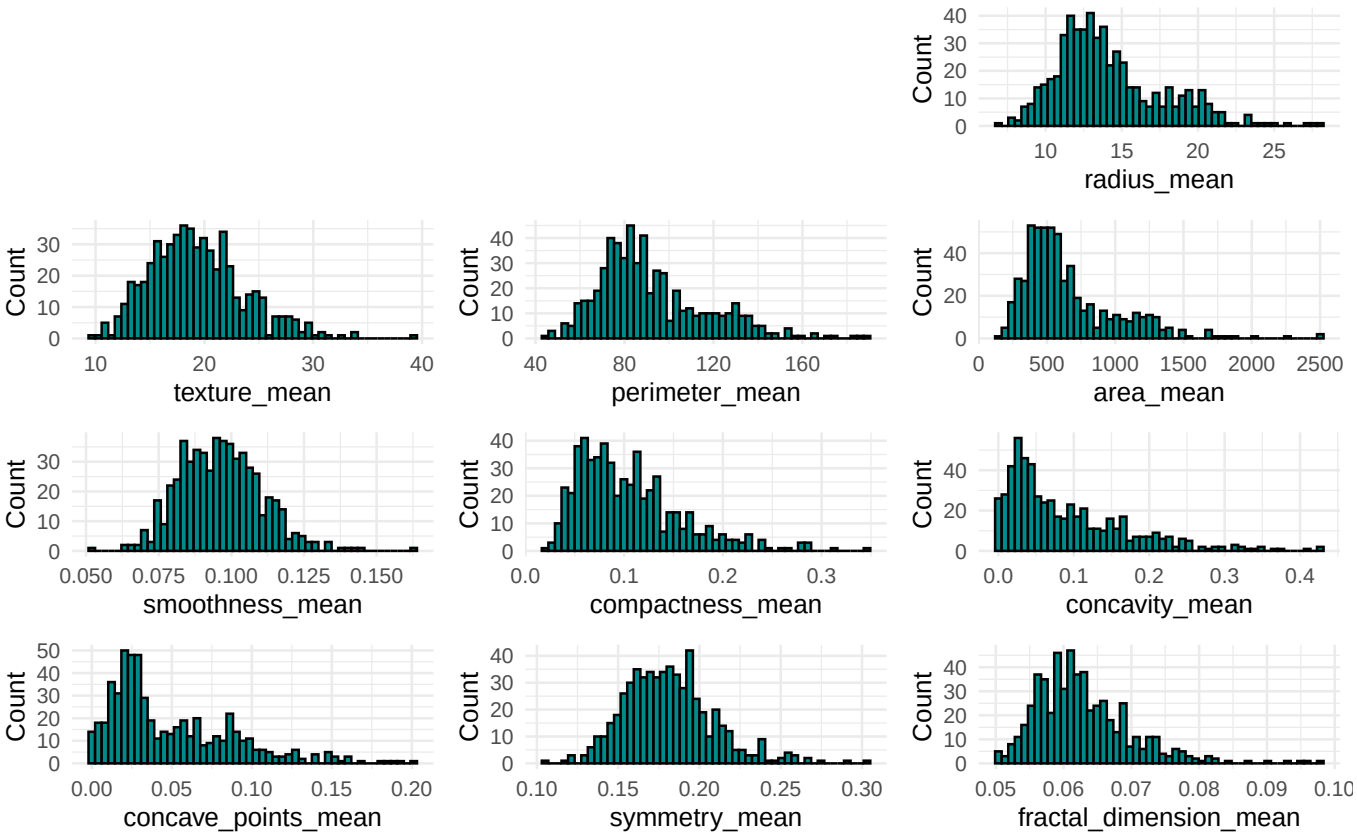

Figure 11: Histograms showing distribution of 'mean' independent variables in the WDBC dataset

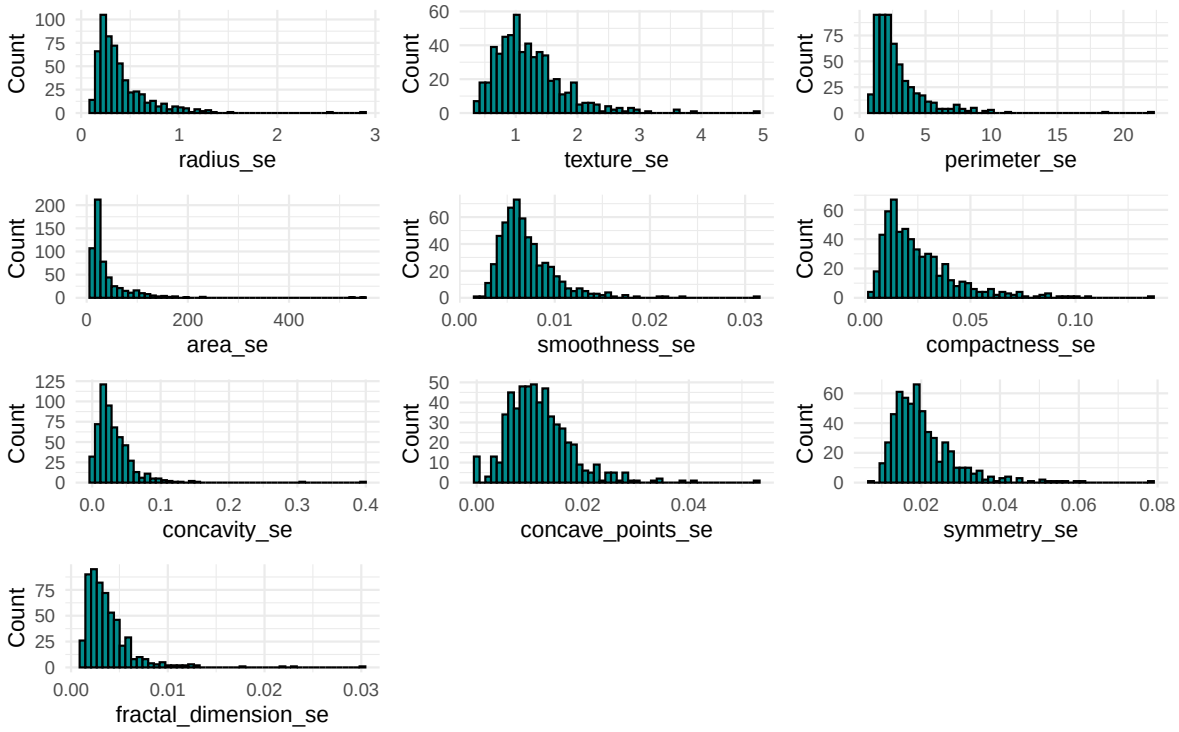

Figure 12: Histograms showing distribution of 'se' independent variables in the WDBC dataset

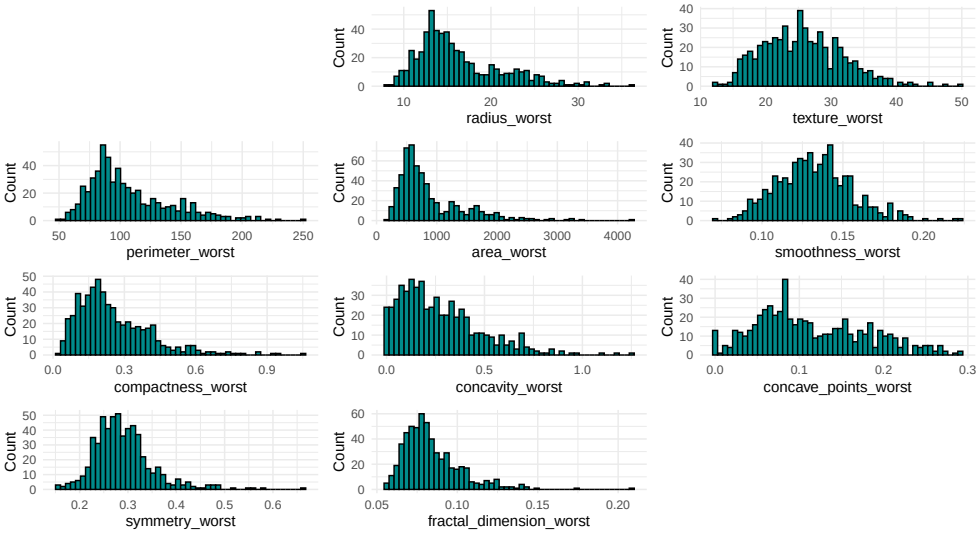

Figure 13: Histograms showing distribution of 'worst' independent variables in the WDBC dataset

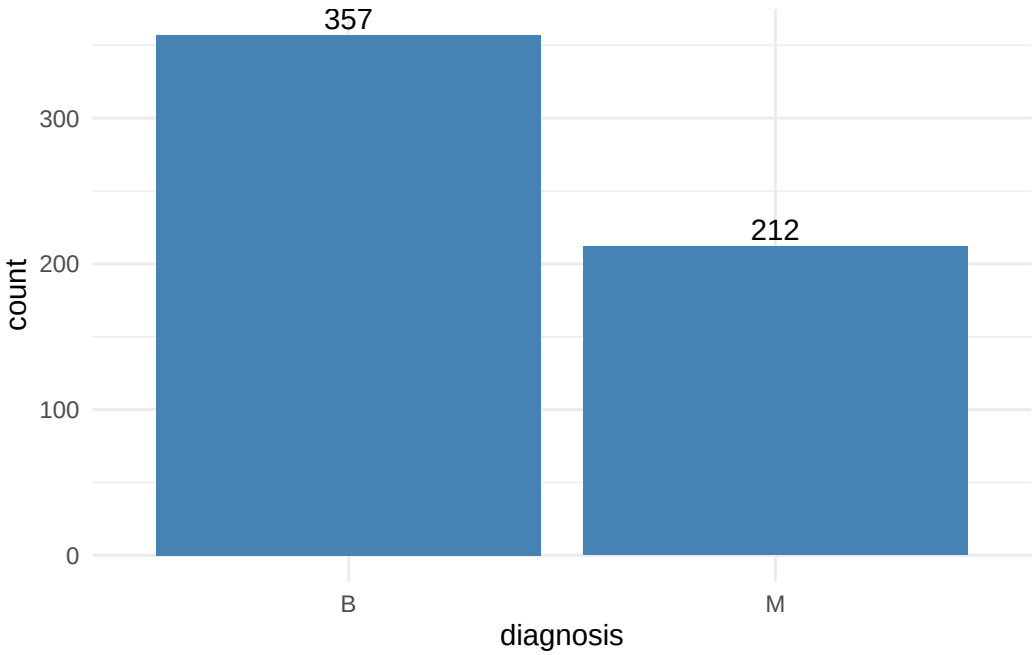

Figure 14: Barplot showing frequency of observations in each class of the dependent variable (diagnosis) in the WDBC dataset

Table 12: Confusion matrix showing predictions made on the original, high-dimensional WDBC test data, where the rows represent the predictions and the columns represent the observed values

	B	M
B	100	5
M	7	58

9 Appendix B

9.1 CG Dataset

Table 13: Head of the CG dataset for class C, with pixel values 1 to 5 (columns) and images 1 to 6 (rows)

pixel 1	pixel 2	pixel 3	pixel 4	pixel 5
0	0	0	0.000	0.000
1	1	1	1.000	1.000
0	0	0	0.000	0.000
0	0	0	0.000	0.008
0	0	0	0.000	0.008
0	0	0	0.008	0.000

Table 14: Head of the CG dataset for class G, with pixel values 1 to 5 (columns) and images 1 to 6 (rows)

pixel 1	pixel 2	pixel 3	pixel 4	pixel 5
0	0	0	0	0.000
1	1	1	1	1.000
0	0	0	0	0.000
0	0	0	0	0.008
0	0	0	0	0.000
0	0	0	0	0.004

Table 15: Tail of the last 6 images (rows) in the CG dataset for class C, with pixel values 780 to 784 (columns)

	pixel 780	pixel 781	pixel 782	pixel 783	pixel 784
1868	0.000	0.000	0.000	0.000	0
1869	0.000	0.000	0.000	0.000	0
1870	0.000	0.000	0.000	0.000	0
1871	0.129	0.106	0.086	0.024	0
1872	0.000	0.000	0.000	0.000	0
1873	0.000	0.000	0.000	0.000	0

Table 16: Tail of the last 6 images (rows) in the CG dataset for class G, with pixel values 780 to 784 (columns)

	pixel 780	pixel 781	pixel 782	pixel 783	pixel 784
1867	0.004	0.004	0.000	0.000	0
1868	0.000	0.000	0.000	0.000	0
1869	0.000	0.000	0.000	0.000	0
1870	0.004	0.059	0.173	0.059	0
1871	0.000	0.000	0.000	0.000	0
1872	0.000	0.000	0.000	0.000	0

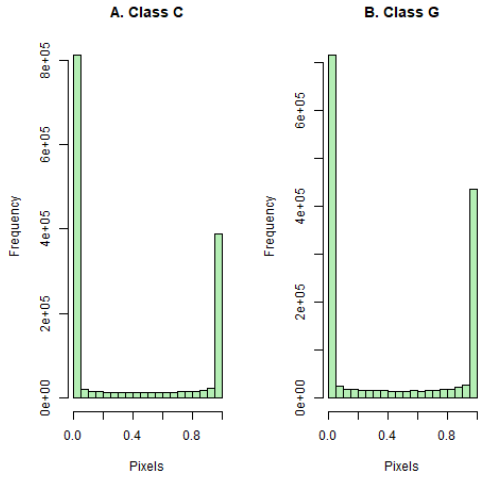

Figure 15: Histograms showing the frequency of pixel values in each class of the CG dataset

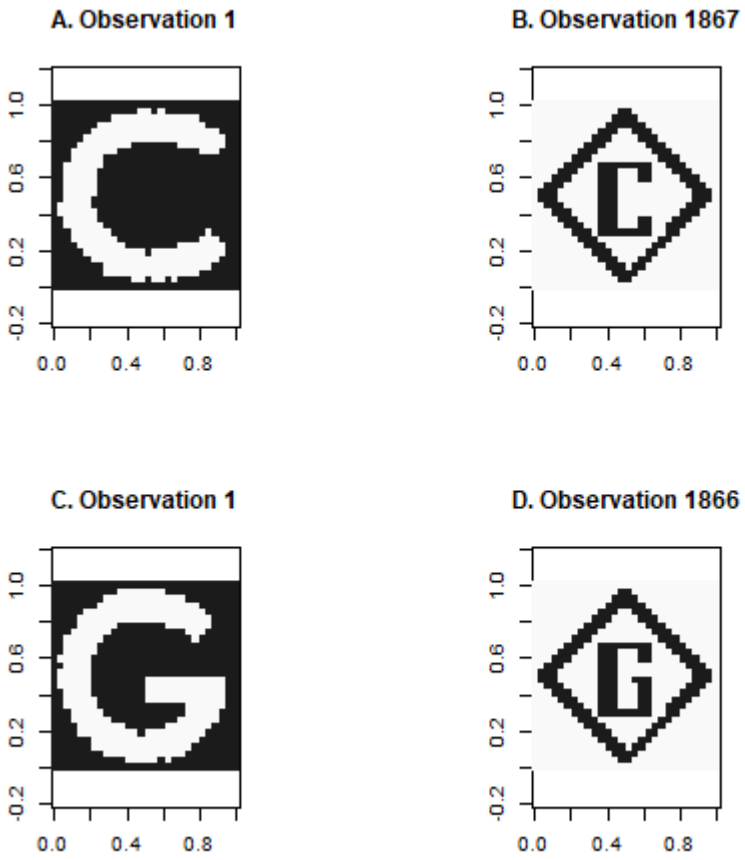

Figure 16: A visual representation of pixel data in the CG dataset

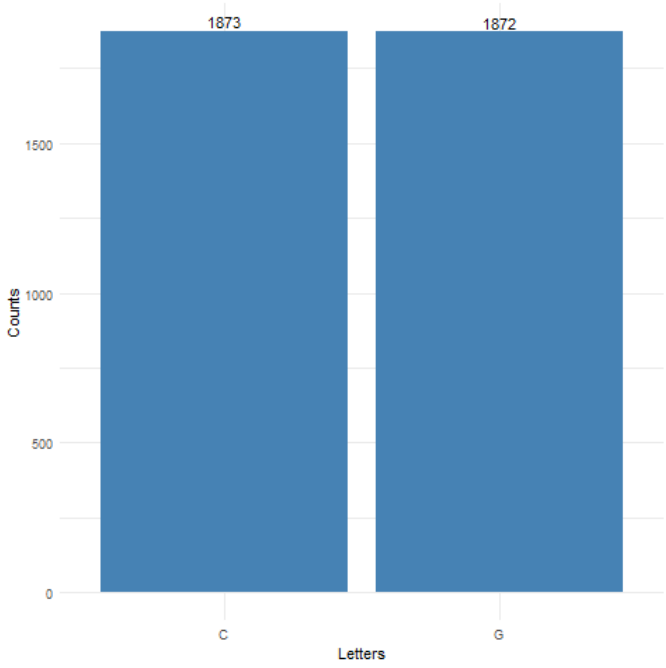

Figure 17: Barplot showing frequency of observations in each class of the dependent variable (letters) in the CG dataset

Table 17: Confusion matrix showing predictions made on the original, high-dimensional CG test data, where the rows represent the predictions and the columns represent the observed values

	c	g
c	542	56
g	19	505

10 Appendix C

10.1 Code

```
# #UNCOMMENT BELOW IF KERAS NEURAL NETWORK MODELS WILL BE BUILT
# #OTHERWISE READ THE SAVED RDS AND RDATA OBJECTS IN EACH CHUNK
#
# # #install remotes
# # install.packages("remotes")
# # remotes::install_github(sprintf("rstudio/%s",c("reticulate", "tensorflow", "keras")))
#
#
# # #Load the reticulate package
library(reticulate)
#
# # #Activate the conda environment in R-studio
use_condaenv('tf-keras') #RENAME to your environment
#
# # #Load the tensorflow library
#library(tensorflow)
#
# # # Install tensorflow in the rminiconda environment
# #install_tensorflow(envname = 'rminiconda') #RENAME to your environment
#
# # #Check if tensorflow is active
#tf$constant("Hello Tensorflow")
#
# # #load library keras
library(keras)


##############################

#GENERAL LIBRARIES

library(dplyr)
library(tidyverse)
library(tidyr)
library(ggplot2)
library(gridExtra)
library(stringr)
library(knitr)
library(kableExtra)
library(corrplot)
library(caret)
```

```
library(png)
library(grid)
library(kohonen)
library(kernlab)


#read in the data
data.wdbc = read.csv("data/WDBC.csv")

#########

#check dimensions of data
dim(data.wdbc)
ncol(data.wdbc)
nrow(data.wdbc)

#########

#look at column names
colnames(data.wdbc)

#rename column names by removing underscore
colnames(data.wdbc) =  c("id", "diagnosis", "radius mean",
                         "texture mean", "perimeter mean",
                         "area mean", "smoothness mean",
                         "compactness mean", "concavity mean",
                         "concave points mean", "symmetry mean",
                         "fractal dimension mean", "radius se",
                         "texture se", "perimeter se", "area se",
                         "smoothness se", "compactness se",
                         "concavity se", "concave points se",
                         "symmetry se", "fractal dimension se",
                         "radius worst", "texture worst",
                         "perimeter worst", "area worst","smoothness worst",
                        "compactness worst", "concavity worst",
                         "concave points worst", "symmetry worst",
                         "fractal dimension worst")   #wdbc data

#########

#look at rownames
rownames(data.wdbc)[1:5]

#########
```

```r
#look at head of data
head.wdbc = head(data.wdbc)

#########

#look at tail of data
tail.wdbc = tail(data.wdbc)

#########

#look at data types
str(data.wdbc)

#convert character and discrete variables to factors
data.wdbc$diagnosis = as.factor(data.wdbc$diagnosis)
data.wdbc$id = as.factor(data.wdbc$id)

#########

#check for missing data
which(complete.cases(data.wdbc) == FALSE)


#summary data
summary.wdbc = data.wdbc %>% select(-id, -diagnosis) %>%
  apply(2, summary) %>% round(3)

#transpose summary data frame to be more readable
summary.wdbc = as.data.frame(t(summary.wdbc))

#add metrics column names
metrics = c("Min", "Q1", "Median","Mean", "Q3", "Max")
colnames(summary.wdbc) = metrics

#get num. indep. variable names
`Numeric independent variables` = data.wdbc %>%
  select(-id, -diagnosis) %>%
  colnames()

#remove rownames
rownames(summary.wdbc) = NULL

#add range column
summary.wdbc = summary.wdbc %>% mutate(Range = Max - Min)
```

```r
#add IQR column
summary.wdbc = summary.wdbc %>% mutate(IQR = Q3-Q1)

#add column with numeric indep. variable names
summary.wdbc = summary.wdbc %>%
  mutate(`Numeric independent variables`, .before=Min)


summary.wdbc %>%
  kable(caption = "Summary statistics of the WDBC dataset",
        format="markdown")


# Assuming data.wdbc is your data frame
data <- data.wdbc

# Replace spaces in column names with underscores
col_names <- colnames(data)
colnames(data) <-  gsub(" ", "_", col_names)

# Get column names
col_names <- colnames(data)

# Create an empty list to store plots
plot_list <- list()

# Loop through each column
for(i in 3:12){
  # Create a histogram for each column
  p <- ggplot(data, aes(x = .data[[col_names[i]]])) +
       geom_histogram(fill = "darkcyan", color = "black", bins=50) +
       labs(x = col_names[i], y = "Count") +
       theme_minimal()

  # Add the plot to the list
  plot_list[[i]] <- p
}


#save image data
saveRDS(plot_list, "images/hist_of_WDBC_mean_variables.rds")


# Create an empty list to store plots
plot_list <- list()
```

```r
# Loop through each column
for(i in 13:22){
  # Create a histogram for each column
  p <- ggplot(data, aes(x = .data[[col_names[i]]])) +
       geom_histogram(fill = "darkcyan", color = "black", bins=50) +
       labs(x = col_names[i], y = "Count") +
       theme_minimal()

  # Add the plot to the list
  plot_list[[i]] <- p
}


#save image data
saveRDS(plot_list, "images/hist_of_WDBC_se_variables.rds")



# Create an empty list to store plots
plot_list <- list()

# Loop through each column
for(i in 23:32){
  # Create a histogram for each column
  p <- ggplot(data, aes(x = .data[[col_names[i]]])) +
       geom_histogram(fill = "darkcyan", color = "black", bins=50) +
       labs(x = col_names[i], y = "Count") +
       theme_minimal()

  # Add the plot to the list
  plot_list[[i]] <- p
}


#save image data
saveRDS(plot_list, "images/hist_of_WDBC_worst_variables.rds")


#calculate counts for each class of diagnosis
counts <- data.wdbc %>%
  group_by(diagnosis) %>%
  summarise(count = n())

#save image data
```

```
saveRDS(counts, "images/barplot_of_WDBC_class.rds")
#correlation matrix
cor.wdbc = data.wdbc %>% select(-id, -diagnosis) %>% #remove id and dep. variable
  cor(method="pearson")

#Correlation plot
plot.cor.wdbc = (corrplot::corrplot(cor.wdbc, tl.cex=0.7, type="upper",
                                    method="color"))


#Which redundant variables should be removed?
set.seed(123)
highlyCorrelated <- caret::findCorrelation(cor.wdbc,
                                           cutoff=0.9,
                                           names=T)
#copy of original data
original.data.wdbc = data.wdbc

#remove variables
data.wdbc = data.wdbc %>% select(-all_of(highlyCorrelated))

#number of indep variables left
dim(data.wdbc) - 2 #exclude diagnosis and ID variable


#stratified sampling into 70% training and 30% test sets
set.seed(123)
wdbc.train.ind <- createDataPartition(data.wdbc$diagnosis, p = .7, list = FALSE, times = 1)

#70% training data
wdbc.train <- data.wdbc[wdbc.train.ind,]

#30% test data
wdbc.test <- data.wdbc[-wdbc.train.ind,]

#check for class imbalance
imbal.wdbc.train = table(wdbc.train$diagnosis)
imbal.wdbc.test = table(wdbc.test$diagnosis)

#upsample training data
set.seed(123)
wdbc.train = upSample(x = wdbc.train[, colnames(wdbc.train) != "diagnosis"],
                      y = as.factor(wdbc.train$diagnosis),
                      yname = "diagnosis")
```

```
#shuffle data
shuffled.ind = sample(nrow(wdbc.train))
wdbc.train = wdbc.train[shuffled.ind, ]


#check that data is balanced
bal.wdbc.train = table(wdbc.train$diagnosis)


#WDBC x-train data
wdbc.train.x = wdbc.train %>% select(-id, -diagnosis)

#WDBC y-train data
wdbc.train.y = wdbc.train %>% select(diagnosis)

#WDBC train IDs
wdbc.train.ids = wdbc.train %>% select(id)

#scale x-training data
wdbc.xtrain.scaled <- scale(wdbc.train.x)

#confirm means of training data are now 0
check.train.mean = round(apply(wdbc.xtrain.scaled, 2, mean), 3)

#confirm std devs of training data are 1
check.train.sd = round(apply(wdbc.xtrain.scaled, 2, sd), 3)

#####################################
#WDBC x-test data
wdbc.test.x = wdbc.test %>% select(-id, -diagnosis)

#WDBC y-test data
wdbc.test.y = wdbc.test %>% select(diagnosis)

#WDBC test IDs
wdbc.test.ids = wdbc.test %>% select(id)

#Scale test data based on training data means and std devs
wdbc.xtest.scaled <- scale(wdbc.test.x,
                           center = attr(wdbc.xtrain.scaled,
                                         "scaled:center"),
                    scale = attr(wdbc.xtrain.scaled,
                                 "scaled:scale"))
```

```
#Means of test x-data close to 0
check.test.mean = round(apply(wdbc.xtest.scaled, 2, mean), 3)

#Standard deviations of test x-data close to 1
check.test.sd = round(apply(wdbc.xtest.scaled, 2, sd), 3)




#10-fold cross validation
trControl <- trainControl(method  = "cv",
                          number  = 10,
                          savePredictions = "all")


#train KNN on original training data
#apply 10 fold cross validation
set.seed(123)
knn.wdbc.original.fit <- caret::train(diagnosis ~ .,
             method     = "knn",
             tuneGrid   = expand.grid(k = 1:100),
             trControl  = trControl,
             metric     = "Accuracy",
             data       = data.frame(diagnosis = wdbc.train.y,
            wdbc.xtrain.scaled))

#CV accuracy
cv.accuracy = knn.wdbc.original.fit

#best value of K
best.k = knn.wdbc.original.fit$bestTune$k


#plot CV accuracy
plot(knn.wdbc.original.fit,
     xlab = "Number of neighbours (K)",
     ylab = "Mean cross-validation accuracy")


#create dataframe with test accuracies
wdbc.test.accuracies = data.frame(matrix(ncol=3, nrow = 26))

#assign column names
colnames(wdbc.test.accuracies) = c("Test Accuracy",
```

```
                                ”Model ID”,
                                ”Model Parameters”)

#add rownames as a column
rownames(wdbc.test.accuracies) = c(”Original”,
                                ”PCA2”,
                                ”PCA3”,
                                ”PCA4”,
                                ”PCA5”,
                                ”PCA6”,
                                ”KPCA2.rbf”,
                                ”KPCA3.rbf”,
                                ”KPCA4.rbf”,
                                ”KPCA5.rbf”,
                                ”KPCA6.rbf”,
                                ”KPCA2.poly”,
                                ”KPCA3.poly”,
                                ”KPCA4.poly”,
                                ”KPCA5.poly”,
                                ”KPCA6.poly”,
                                ”Autoencoder2”,
                                ”Autoencoder3”,
                                ”Autoencoder4”,
                                ”Autoencoder5”,
                                ”Autoencoder6”,
                                ”SOM2”,
                                ”SOM3”,
                                ”SOM4”,
                                ”SOM5”,
                                ”SOM6”)



#refit KNN model on train data using optimal value of K

set.seed(123)
best.wdbc.original.fit <- knn3(
                diagnosis ~ .,
                data = data.frame(diagnosis = wdbc.train.y,
                                  wdbc.xtrain.scaled),
                k = best.k
               )

#model prediction on test set
predictions <- predict(best.wdbc.original.fit,
```

```
                    data.frame(wdbc.xtest.scaled),
                    type = "class")

#create dataframe of predictions vs observed
colnames(wdbc.test.y) = "observed" #change column name
wdbc.true.vs.pred = wdbc.test.y %>% mutate(predictions)

#test confusion matrix
wdbc.test.metrics <- confusionMatrix(wdbc.true.vs.pred$predictions,
                                     wdbc.true.vs.pred$observed)

#assign test accuracy to dataframe
wdbc.test.accuracies["Original", "Test Accuracy"] = round(wdbc.test.metrics$overall["Accuracy"], 3)

#assign model ID
wdbc.test.accuracies["Original", "Model ID"] = "wdbc.original.K7"

#assign model parameters
wdbc.test.accuracies["Original", "Model Parameters"] = paste0("K = ",
                                    knn.wdbc.original.fit$bestTune$k)

#Extract confusion matrix
wdbc.original.conf.mat = wdbc.test.metrics$table


#apply pca using prcomp on scaled x data
pca.wdbc.fit = prcomp(wdbc.xtrain.scaled)

#summary of pca analysis
summary.pca <- summary(pca.wdbc.fit)

#get eigenvalues or variance of each principal component
prcomp.variance <- pca.wdbc.fit$sdev^2

#put variance values in a dataframe
prcomp.variance = data.frame("PC"=1:20,
                             "Eigenvalues"= prcomp.variance)

############

# plot variance of principal component
p1 <- ggplot(prcomp.variance, aes( x = PC, y = Eigenvalues) ) +
  geom_line() +
  geom_hline( yintercept = 0.01, linetype = 'dotted', col = 'red') +
  theme_minimal() +
```

```
  xlab( 'Principal Component Number' ) +
  ylab( 'Eigenvalues' ) +
  ggtitle( 'A.' )

###############

#prcomp: proportion of variance
prcomp.variance$proportion.of.variance <- prcomp.variance$Eigenvalues/sum(prcomp.variance$Eigenvalues)

# plot proportion of variance of each principal component
p2 <- ggplot(prcomp.variance, aes( x = PC, y = proportion.of.variance) ) +
  geom_line() +
  geom_hline( yintercept = 0.01, linetype = 'dotted', col = 'red') +
  theme_minimal() +
  xlab( 'Principal Component Number' ) +
  ylab( 'Prop. Explained Var.' ) +
  ggtitle( 'B.' )

###############

#prcomp: cumulative proportion of variance
prcomp.variance$cum.variance <- cumsum(prcomp.variance$proportion.of.variance)

# plot cumulative variance
p3 <- ggplot(prcomp.variance, aes( x = PC, y = cum.variance) ) +
  geom_line() +
  geom_hline( yintercept = 0.95, linetype = 'dotted', col = 'red') +
  theme_minimal() +
  xlab( 'Principal Component Number' ) +
  ylab( 'Cum. Prop. Explained Var.' ) +
  ggtitle( 'C.' )

#grid plot
grid.arrange( p1, p2, p3,
              ncol=2)



#dimensions to assess
dimensions = c(2, 3, 4, 5, 6)

#rownames
PCA.dims = c("PCA2", "PCA3", "PCA4", "PCA5", "PCA6")
```

```
#validation accuracies
valid.accuracy = c()

####################

for (dim in dimensions) {

  #train scores
  pca.wdbc.train.scores = pca.wdbc.fit$x[,1:dim]

  ####################

  #test scores
  pca.wdbc.test.scores <- data.frame(predict(pca.wdbc.fit,
                          newdata = wdbc.xtest.scaled))

  #choose dimensions for test set
  pca.wdbc.test.scores = pca.wdbc.test.scores[,1:dim]

  ####################
  #Find optimal value of K

  #10-fold cross validation
  trControl <- trainControl(method  = "cv",
                            number  = 10,
                            savePredictions = "all")

  #apply 10 fold cross validation
  set.seed(123)
  knn.wdbc.pca.fit <- caret::train(diagnosis ~ .,
              method     = "knn",
              tuneGrid   = expand.grid(k = 1:100),
              trControl  = trControl,
              metric     = "Accuracy",
              data       = data.frame(diagnosis = wdbc.train.y,
             pca.wdbc.train.scores))

  #CV accuracy
  cv.accuracy = knn.wdbc.pca.fit

  #best value of K
  best.k = knn.wdbc.pca.fit$bestTune$k

  #validation accuracy
  valid.accuracy = cbind(valid.accuracy,
```

```
                    cv.accuracy$results$Accuracy)


  ###################
  #refit KNN model on train data using optimal value of K
  set.seed(123)
  best.wdbc.pca.fit <- knn3(
                  diagnosis ~ .,
                  data = data.frame(diagnosis = wdbc.train.y,
                                    pca.wdbc.train.scores),
                  k = best.k
                 )

  ####################
  #model prediction on test set
  predictions <- predict(best.wdbc.pca.fit,
                     data.frame(pca.wdbc.test.scores),
                     type = "class")

  #create dataframe of predictions vs observed
  colnames(wdbc.test.y) = "observed" #change column name
  wdbc.true.vs.pred = wdbc.test.y %>% mutate(predictions)

  #test confusion matrix
  wdbc.test.metrics <- confusionMatrix(wdbc.true.vs.pred$predictions,
                                       wdbc.true.vs.pred$observed)

  ######################
  #assign test accuracy to dataframe
  wdbc.test.accuracies[PCA.dims[dim-1], "Test Accuracy"] = round(wdbc.test.metrics$overall["Accuracy"], 3)

  #assign model ID
  wdbc.test.accuracies[PCA.dims[dim-1], "Model ID"] = paste0("wdbc.pca.K",
                                                             best.k,
                                                             ".D", dim)

  #assign model parameters
  wdbc.test.accuracies[PCA.dims[dim-1], "Model Parameters"] = paste0("K = ",
                                    best.k)

}


#convert validation accuracies to dataframe
valid.accuracy = data.frame(valid.accuracy)
```

```
#column names
colnames(valid.accuracy) = PCA.dims

#plot mean cv accuracies
ggplot(valid.accuracy,
       aes(x = 1:100)) +
  geom_line(aes(y = PCA2, color = "PCA2"), linetype = "solid") +
  geom_line(aes(y = PCA3, color = "PCA3"), linetype = "solid") +
  geom_line(aes(y = PCA4, color = "PCA4"), linetype = "solid") +
  geom_line(aes(y = PCA5, color = "PCA5"), linetype = "solid") +
  geom_line(aes(y = PCA6, color = "PCA6"), linetype = "solid") +
  labs(title = "",
       x = "Number of neighbours (K)",
       y = "Mean CV accuracy",
       color = "Legend")+
    theme_minimal() +
  scale_color_manual(values = c("darkblue", "darkred", "darkgreen", "darkorange", "black"),
                     labels = c("PCA2", "PCA3", "PCA4", "PCA5", "PCA6"))



#dimensions to assess
dimensions = c(2, 3, 4, 5, 6)

#Kernel functions
Kernels = c("rbfdot","polydot")

#rownames
KPCA.dims = c("KPCA2.rbf", "KPCA3.rbf", "KPCA4.rbf", "KPCA5.rbf", "KPCA6.rbf",
              "KPCA2.poly", "KPCA3.poly", "KPCA4.poly", "KPCA5.poly", "KPCA6.poly")

#validation accuracies
valid.accuracy = c()

#best.k values
all.best.k = c()

# Define training control
train.control <- trainControl(method = "cv", number = 10)

####################

for (dim in dimensions) {
```

```
for (kern in Kernels) {


  if (kern == "rbfdot") {
    set.seed(123)
        kpca.wdbc.fit <- kpca(~.,
              data = data.frame(wdbc.xtrain.scaled),
              kernel = kern,
              kpar = list(sigma = 0.1),
              features = dim)

} else { #polydot
      set.seed(123)
      kpca.wdbc.fit <- kpca(~.,
              data = data.frame(wdbc.xtrain.scaled),
              kernel = kern,
              kpar = list(degree = 1,
                          scale = 1, offset = 1),
              features = dim)
}

  #train scores
  kpca.wdbc.train.scores = kpca.wdbc.fit@pcv

  ######################

  #test scores
  kpca.wdbc.test.scores <- predict(kpca.wdbc.fit,
                           data.frame(wdbc.xtest.scaled))
  ######################

  #Find optimal value of K

  #10-fold cross validation
  trControl <- trainControl(method  = "cv",
                            number  = 10,
                            savePredictions = "all")

  #apply 10 fold cross validation
  set.seed(123)
  knn.wdbc.kpca.fit <- caret::train(diagnosis ~ .,
                method     = "knn",
                tuneGrid   = expand.grid(k = 1:100),
                trControl  = trControl,
                metric     = "Accuracy",
```

```
            data       = data.frame(diagnosis = wdbc.train.y,
          kpca.wdbc.train.scores))

#CV accuracy
cv.accuracy = knn.wdbc.kpca.fit

#best value of K
best.k = knn.wdbc.kpca.fit$bestTune$k

#validation accuracy
valid.accuracy = cbind(valid.accuracy,
                  cv.accuracy$results$Accuracy)


###################
#refit KNN model on train data using optimal value of K
set.seed(123)
best.wdbc.kpca.fit <- knn3(
              diagnosis ~ .,
              data = data.frame(diagnosis = wdbc.train.y,
                          kpca.wdbc.train.scores),
              k = best.k
             )

####################

#model prediction on test set
predictions <- predict(best.wdbc.kpca.fit,
                 data.frame(kpca.wdbc.test.scores),
                 type = "class")

#create dataframe of predictions vs observed
colnames(wdbc.test.y) = "observed" #change column name
wdbc.true.vs.pred = wdbc.test.y %>% mutate(predictions)

#test confusion matrix
wdbc.test.metrics <- confusionMatrix(wdbc.true.vs.pred$predictions,
                                     wdbc.true.vs.pred$observed)

######################

#offset for each Kernel function - used to populate test accuracy table
if (kern == Kernels[1]) {
  offset = 0
  sigma = 0.1
```

```
  degree = NA #HERE
  scale = NA #here
  offset.model.param = NA #here
} else if (kern == Kernels[2]) {
  offset = +5
  sigma = NA
  degree = 1
  scale = 1 #here
  offset.model.param = 1 #here

}

#assign test accuracy to dataframe
wdbc.test.accuracies[KPCA.dims[dim-1+offset],
                     "Test Accuracy"] = round(wdbc.test.metrics$overall["Accuracy"], 3)

#assign model ID
wdbc.test.accuracies[KPCA.dims[dim-1+offset], #-1 because dims start at 2
                     "Model ID"] = paste0("wdbc.kpca.K",
                                          best.k, ".",
                                          kern,
                                          ".D",
                                          dim)

#assign model parameters
wdbc.test.accuracies[KPCA.dims[dim-1+offset],
                     "Model Parameters"] = paste0("K = ",
                                                  best.k,
                                                  ", Kernel = ",
                                                  kern,

                                                  ifelse(!is.na(sigma),
                                                         paste0(", sigma = ",
                                                         sigma), ""),

                                                  ifelse(!is.na(degree), #here
                                                         paste0(", degree = ",
                                                         degree), ""),


                                                   ifelse(!is.na(offset), #here
                                                         paste0(", offset = ",
                                                         offset.model.param), ""),
```

```
                                ifelse(!is.na(scale), #here
                                       paste0(", scale = ",
                                       scale), "")
                                )
  }
}


#convert validation accuracies to dataframe
valid.accuracy = data.frame(valid.accuracy)

#column names
colnames(valid.accuracy) =
  c("KPCA2.rbfdot",
    "KPCA2.polydot",
    "KPCA3.rbfdot",
    "KPCA3.polydot",
    "KPCA4.rbfdot",
    "KPCA4.polydot",
    "KPCA5.rbfdot",
    "KPCA5.polydot",
    "KPCA6.rbfdot",
    "KPCA6.polydot")


#plot mean cv accuracies
ggplot(valid.accuracy,
       aes(x = 1:100)) +
  geom_line(aes(y = KPCA2.rbfdot,
                color = "KPCA2.rbfdot"),
            linetype = "solid") +
  geom_line(aes(y = KPCA2.polydot,
                color = "KPCA2.polydot"),
            linetype = "solid") +
  geom_line(aes(y = KPCA3.rbfdot,
                color = "KPCA3.rbfdot"),
            linetype = "solid") +
  geom_line(aes(y = KPCA3.polydot,
                color = "KPCA3.polydot"),
            linetype = "solid") +
  geom_line(aes(y = KPCA4.rbfdot,
                color = "KPCA4.rbfdot"),
            linetype = "solid") +
  geom_line(aes(y = KPCA4.polydot,
                color = "KPCA4.polydot"),
```

```
            linetype = "solid") +
  geom_line(aes(y = KPCA5.rbfdot,
                color = "KPCA5.rbfdot"),
            linetype = "solid") +
  geom_line(aes(y = KPCA5.polydot,
                color = "KPCA5.polydot"),
            linetype = "solid") +
  geom_line(aes(y = KPCA6.rbfdot,
                color = "KPCA6.rbfdot"),
            linetype = "solid") +
  geom_line(aes(y = KPCA6.polydot,
                color = "KPCA6.polydot"),
            linetype = "solid") +
  labs(title = "",
       x = "Number of neighbours (K)",
       y = "Mean CV accuracy",
       color = "Legend")+
    theme_minimal() +
scale_color_brewer(palette = "Paired") +
  guides(color = guide_legend(override.aes = list(size = 2)))



##############################
#ensure data is a matric
check = is.matrix(wdbc.xtrain.scaled)

#ensure data is numeric
check = is.numeric(wdbc.xtrain.scaled)

#####################################

#dimensions to assess
dimensions = c(2, 3, 4, 5, 6)

#rownames
SOM.dims = c("SOM2", "SOM3", "SOM4", "SOM5", "SOM6")

#validation accuracies
valid.accuracy = c()

# Define training control
train.control <- trainControl(method = "cv", number = 10)

#####################################
```

```
#for loop to:
 ##train SOM model on training data, where SOM model:
  ### is trained on training data
  ###a given dim x dim dimensions

 ##Make predictions on test and training set to get reduced features
 ##build knn model on reduced training data and apply 10 fold CV
 ## determine optimal K with largest CV accuracy
 ##rebuild knn model on reduced train data using optimal K
 ##predict test data label using reduced test data and knn model
 ##determine test data accuracy by assessing confusion matrix

#topologies
topologies = c("hexagonal", "rectangular")

#best cv accuracy given the best k
##results given per topology
best.k.cv.accuracy = data.frame(hexagonal = double(),
                                rectangular = double())

#for each toplogy and dimension combination...
for (topo in topologies) {
  for (dim in dimensions) {

    #SOM grid
    som.grid <- kohonen::somgrid(xdim = dim,
                         ydim=dim,
                         topo=topo) #hexagonal or rectangular

    #build SOM model
    set.seed(123)
    som.model <- kohonen::som(wdbc.xtrain.scaled,
                      grid=som.grid,
                      rlen=500, #number of iterations
                      alpha=c(0.05,0.01), #learning rate
                      keep.data = TRUE
                      )

    #predict train reduced data
    set.seed(123)
    som.reduced.train = predict(som.model, wdbc.xtrain.scaled)$predictions[[1]]

    #10-fold cross validation
    trControl <- trainControl(method  = "cv",
```

```
                        number  = 10,
                        savePredictions = "all")

    #Apply KNN on SOM model with given dimension and topology:
    #apply 10 fold cross validation
    set.seed(123)
    knn.wdbc.som.fit <- train(diagnosis ~ .,
                      method     = "knn",
                      tuneGrid   = expand.grid(k = 1:100), #K grid search
                      trControl  = trControl,
                      metric     = "Accuracy",
                      data       = data.frame(diagnosis = wdbc.train.y,
                    som.reduced.train)) #take in SOM reduced train data

    #obtain CV accuracies for each value of k
    cv.accuracy = knn.wdbc.som.fit

    #best value of K with largest cv accuracy
    best.k = knn.wdbc.som.fit$bestTune$k

    #extract largest valid accuracy given best k
    #append to dataframe
    best.k.cv.accuracy[dim-1, topo] = cv.accuracy$results$Accuracy[best.k]
  }
}

##################################

#for each dimension 2 to 6
for (dim in dimensions) {

  #Find best topology with greated CV accuracy
  #topology = hexagonal if CV accuracy for hexagonal>= rectangular
  if (best.k.cv.accuracy[dim-1, "hexagonal"] >= best.k.cv.accuracy[dim-1, "rectangular"]) { topo = "hexagonal"
  } else {
    topo = "rectangular"
  }

  #SOM grid
  som.grid <- kohonen::somgrid(xdim = dim,
                       ydim=dim,
                       topo=topo) #best topology

  #build SOM model
  set.seed(123)
```

```
som.model <- kohonen::som(wdbc.xtrain.scaled,
                    grid=som.grid,
                    rlen=500,
                    alpha=c(0.05,0.01),
                    keep.data = TRUE
                    )

#predict train reduced data
set.seed(123)
som.reduced.train = predict(som.model, wdbc.xtrain.scaled)$predictions[[1]]

#predict test reduced data
set.seed(123)
som.reduced.test = predict(som.model, wdbc.xtest.scaled)$predictions[[1]]

#Find optimal value of K

#10-fold cross validation
trControl <- trainControl(method  = "cv",
                          number  = 10,
                          savePredictions = "all")

#Apply KNN on SOM model with given dimension and topology:
#apply 10 fold cross validation
set.seed(123)
knn.wdbc.som.fit <- train(diagnosis ~ .,
                    method     = "knn",
                    tuneGrid   = expand.grid(k = 1:100), #K grid search
                    trControl  = trControl,
                    metric     = "Accuracy",
                    data       = data.frame(diagnosis = wdbc.train.y,
                   som.reduced.train)) #take in SOM reduced train data

#obtain CV accuracies for each value of k
cv.accuracy = knn.wdbc.som.fit

#best value of K with largest cv accuracy
best.k = knn.wdbc.som.fit$bestTune$k

#validation accuracy for each value of k
valid.accuracy = cbind(valid.accuracy,
                       cv.accuracy$results$Accuracy)


##################
```

```
#refit KNN model on train data using optimal value of K
set.seed(123)
best.wdbc.som.fit <- knn3(
                    diagnosis ~ .,
                    data = data.frame(diagnosis = wdbc.train.y,
                                      som.reduced.train),
                    k = best.k
                   )

####################

#model prediction on test set
predictions <- predict(best.wdbc.som.fit,
                       data.frame(som.reduced.test),
                       type = "class")

#create dataframe of predictions vs observed
colnames(wdbc.test.y) = "observed" #change column name
  wdbc.true.vs.pred = wdbc.test.y %>% mutate(predictions)

#test confusion matrix
wdbc.test.metrics <- confusionMatrix(wdbc.true.vs.pred$predictions,
                                     wdbc.true.vs.pred$observed)

  ##########################

#assign test accuracy to dataframe
wdbc.test.accuracies[SOM.dims[dim-1], #-1 because dims start at 2
                     "Test Accuracy"] = round(wdbc.test.metrics$overall["Accuracy"], 3)

#assign model ID
wdbc.test.accuracies[SOM.dims[dim-1], #-1 because dims start at 2
                     "Model ID"] = paste0("wdbc.som.K",
                                          best.k,
                                          ".", substr(topo, 1, 3),
                                          ".D",
                                          dim)

#assign model parameters
wdbc.test.accuracies[SOM.dims[dim-1],
                     "Model Parameters"] = paste0("K = ", best.k,
                                                  ", topology = ", topo,
                                                  ", rlen = 500",
                                                  "alpha = [0.05, 0.01]")
```

```
}

#convert validation accuracies to dataframe
valid.accuracy = data.frame(valid.accuracy)

#column names
colnames(valid.accuracy) = SOM.dims

#plot mean cv accuracies
ggplot(valid.accuracy,
       aes(x = 1:100)) +
    geom_line(aes(y = SOM2,
                  color = "SOM2"),
              linetype = "solid") +
    geom_line(aes(y = SOM3,
                  color = "SOM3"),
              linetype = "solid") +  geom_line(aes(y = SOM4,
                  color = "SOM4"),
              linetype = "solid") +
    geom_line(aes(y = SOM5,
                  color = "SOM5"),
              linetype = "solid") +
    geom_line(aes(y = SOM6,
                  color = "SOM6"),
              linetype = "solid") +

  labs(title = "",
       x = "Number of neighbours (K)",
       y = "Mean CV accuracy",
       color = "Legend")+
    theme_minimal() +
scale_color_brewer(palette = "Paired") +
  guides(color = guide_legend(override.aes = list(size = 8, alpha =2)))




wdbc.autoen.valid.results = readRDS("images/wdbc.autoen.valid.results.rds")

#UNCOMMENT CODE BELOW IF REQUIRED.
#SET UP CONDA ENVIRONMENT!
# ##################################################################
#
# #Set random seed
```

```
# set.seed(123)
# tensorflow::set_random_seed(123)
#
#
# #GRIDSEARCH
# ##Ensure butterfly shape
# input.nodes = dim(wdbc.xtrain.scaled)[2] #also hidden nodes
# hidden.nodes = c(15, 10, 5) #less than input nodes but greater than dimensions
# dimensions = c(2, 3, 4, 5, 6) #bottleneck
# activation.grid = c("relu", "tanh")
#
# #Make the target variable is numeric, starting at 0
# wdbc.train.y.auto <- as.integer(factor(wdbc.train.y$diagnosis)) - 1 #train
# wdbc.test.y.auto <- as.integer(factor(wdbc.test.y$observed)) - 1 #test
#
# #save a copy of original y-data before hot-coding
# wdbc.train.y.auto.original <- wdbc.train.y.auto #train
# wdbc.test.y.auto.original <- wdbc.test.y.auto #test
#
# #hot-coding
# wdbc.train.y.auto <- to_categorical(wdbc.train.y.auto) #train
# wdbc.test.y.auto <- to_categorical(wdbc.test.y.auto) #test
#
# #ensure numeric, matrix feature data
# is.matrix(wdbc.xtrain.scaled) #train
# is.matrix(wdbc.xtest.scaled) #test
#
# ##########################################################################
# #DEFINE NUMBER OF FOLDS
# num.folds <- 10
#
# #Split the data into k-folds
# set.seed(123)
# folds <- createFolds(wdbc.train.y$diagnosis, k = num.folds)
#
# #empty dataframe for validation results
# wdbc.autoen.valid.results = data.frame(valid.accuracy = c(),
#                                        parameters = c())
#
# #PERFORM GRIDSEARCH
# for (hidden.node in hidden.nodes) {
#   for (dim in dimensions) {
#     for (activ in activation.grid) {
#       ################################
#       #empty vectors for loss at each fold...
```

```
#         #accuracy
#         valid.loss.values <- numeric(num.folds) #validation
#
#         ##########################################
#
#         #for each fold...
#         for (fold in 1:num.folds) {
#
#         #extract train and validation folds
#         train.indices <- unname(unlist(folds[-fold]))
#         valid.indices <- unname(unlist(folds[fold]))
#
#         #prepare features at each fold
#         x.train <- wdbc.xtrain.scaled[train.indices, ] #train
#         x.valid <- wdbc.xtrain.scaled[valid.indices, ] #validation
#
#         #prepare target data at each fold
#         y.train <- wdbc.train.y.auto[train.indices,] #train
#         y.valid <- wdbc.train.y.auto[valid.indices,] #validation
#
#         ##########################################
#
#          #fit the model on the training-fold data
#          #with parameters at the current iteration
#
#          #initialize model
#          set.seed(123)
#          autoen.model <- keras_model_sequential()
#
#          #define the model
#          autoen.model%>%
#          layer_dense(units = hidden.node,
#                      activation = activ,
#                      input_shape = dim(x.train)[2]) %>%
#          layer_dense(units = dim, activation = activ, name = "bottleneck")%>%
#          layer_dense(units = hidden.node, activation = activ) %>%
#          layer_dense(units = dim(x.train)[2],
#                      activation = "sigmoid")
#
#          ##compile model
#          autoen.model %>% compile(
#            loss = "mse",
#            optimizer = 'adam',
#           )
#
```

```
#         ##train the model on train-fold data
#         set.seed(123)
#         autoen.model %>% fit(
#           x = x.train,
#           y = x.train,
#           epochs = 25,
#           batch_size=128,
#           verbose = 0)
#       ##########################################
#
#       #Predictions on valid data
#        set.seed(123)
#        valid.metrics <- autoen.model %>% evaluate(x.valid, x.valid)
#
#        ##valid loss
#        valid.loss <- unname(valid.metrics[1])
#
#        #append validation loss for this fold
#         valid.loss.values = c(valid.loss.values,
#                               valid.loss)
#       } #end of 10 folds
#
#       ##########################################
#
#       # Calculate the mean results across all folds
#
#       ##LOSS
#       mean.valid.loss <- mean(valid.loss.values) #validation
#
#       ##########################################
#
#       # Store the results
#       wdbc.autoen.valid.results = rbind(wdbc.autoen.valid.results,
#                                   c(mean.valid.loss,
#                                     paste0("No. of hidden nodes = ",
#                                            hidden.node,
#                                            ". Dimensions = ",
#                                            dim,
#                                            ". Activation function = ",
#                                            activ)))
#     }
#   }
# }

#####################################
```

```
#save results
#saveRDS(wdbc.autoen.valid.results, "images/wdbc.autoen.valid.results.rds")

#####################################

#change column names
colnames(wdbc.autoen.valid.results) = c("Mean CV Loss", "Parameters")

#round off loss to 3 decimal places
wdbc.autoen.valid.results$`Mean CV Loss` = round(as.double(wdbc.autoen.valid.results$`Mean CV Loss`), 3)

#separate hyperparameters into separate columns
wdbc.autoen.valid.results <- separate(wdbc.autoen.valid.results,
                                      "Parameters",
                                      into = c("No. of hidden nodes",
                                               "Dimensions",
                                               "Activation function"),
                                      sep = "\\. Dimensions = |\\. Activation function = ")

#extract number of hidden nodes
wdbc.autoen.valid.results$`No. of hidden nodes` = gsub(".*=\\s*", "",
    wdbc.autoen.valid.results$`No. of hidden nodes`)


#ensure numeric values
##hidden nodes
wdbc.autoen.valid.results$`No. of hidden nodes` = as.integer(wdbc.autoen.valid.results$`No. of hidden nodes`)

##dimensions
wdbc.autoen.valid.results$Dimensions = as.integer(wdbc.autoen.valid.results$Dimensions)

#########################################

#RETRAIN AUTOENCODER WITH OPTIMAL HYPERPARAMETERS

#empty validation vector
valid.accuracy = c()

#AUTOENCODER DIMENSION IDs
autoen.dims = c("Autoencoder2",
                "Autoencoder3",
                "Autoencoder4",
                "Autoencoder5",
                "Autoencoder6")
```

```
for (dim in dimensions) {

  #extract log loss values for specific dimension
  optimal.logloss = wdbc.autoen.valid.results%>%
    filter(Dimensions == dim)

  #find smallest CV reconstruction error
  min.row = which.min(apply(optimal.logloss,
                            1,
                            function(x) x[1]))

  #get parameters for min CV Loss
  hidden.nodes = optimal.logloss[min.row, "No. of hidden nodes"]

  activ = optimal.logloss[min.row, "Activation function"]

  ################################

  #initialize model
  set.seed(123)
  tensorflow::set_random_seed(123)
  autoen.model <- keras_model_sequential()

  #define the model
  autoen.model%>%
    layer_dense(units = hidden.nodes,
                activation = activ,
                input_shape = dim(wdbc.xtrain.scaled)[2]) %>%
    layer_dense(units = dim,
                activation = activ,
                name = "bottleneck")%>%
              layer_dense(units = hidden.nodes,
                          activation = activ) %>%
              layer_dense(units = dim(wdbc.xtrain.scaled)[2],
                          activation = "sigmoid")
  ##compile model
  autoen.model %>% compile(
              loss = "mse",
              optimizer = 'adam',
            )

  ##train the model on train-fold data
  set.seed(123)
```

```
autoen.model %>% fit(
x = wdbc.xtrain.scaled,
y = wdbc.xtrain.scaled,
epochs = 25,
batch_size=128,
verbose = 0)

##########################################
#Reduced train data
 autoen.reduced.train = predict(autoen.model, wdbc.xtrain.scaled)

#column names for train data
colnames(autoen.reduced.train) = colnames(wdbc.xtrain.scaled)

##################################################

#Reduced test data
autoen.reduced.test = predict(autoen.model, wdbc.xtest.scaled)

#column names for test data
colnames(autoen.reduced.test) = colnames(wdbc.xtest.scaled)

##################################################

#BUILD KNN MODEL

#10-fold cross validation
trControl <- trainControl(method  = "cv",
                          number  = 10,
                          savePredictions = "all")

#Apply KNN on SOM model with given dimension and topology:
#apply 10 fold cross validation
set.seed(123)
knn.wdbc.autoen.fit <- caret::train(diagnosis ~ .,
                  method     = "knn",
                  tuneGrid   = expand.grid(k = 1:100), #K grid search
                  trControl  = trControl,
                  metric     = "Accuracy",
                  data       = data.frame(diagnosis = wdbc.train.y,
                autoen.reduced.train)) #autoencoder reduced train data

#obtain CV accuracies for each value of k
cv.accuracy = knn.wdbc.autoen.fit
```

```
#best value of K with largest cv accuracy
best.k = knn.wdbc.autoen.fit$bestTune$k

#validation accuracy for each value of k
valid.accuracy = cbind(valid.accuracy,
                       cv.accuracy$results$Accuracy)

###################
#refit KNN model on train data using optimal value of K
set.seed(123)
best.wdbc.autoen.fit <- knn3(
                     diagnosis ~ .,
                     data = data.frame(diagnosis = wdbc.train.y,
                                       autoen.reduced.train),
                     k = best.k
                    )
#####################
#model prediction on test set
predictions <- predict(best.wdbc.autoen.fit,
                       data.frame(autoen.reduced.test),
                       type = "class")

#create dataframe of predictions vs observed
colnames(wdbc.test.y) = "observed" #change column name
  wdbc.true.vs.pred = wdbc.test.y %>% mutate(predictions)

#test confusion matrix
wdbc.test.metrics <- confusionMatrix(wdbc.true.vs.pred$predictions,
                                     wdbc.true.vs.pred$observed)

#assign test accuracy to dataframe
wdbc.test.accuracies[autoen.dims[dim-1], #-1 because dims start at 2
                     "Test Accuracy"] = round(wdbc.test.metrics$overall["Accuracy"], 3)

#assign model ID
wdbc.test.accuracies[autoen.dims[dim-1], #-1 because dims start at 2
                     "Model ID"] = paste0("wdbc.autoen.K",
                                          best.k,
                                          ".D",
                                          dim)

#assign model parameters
wdbc.test.accuracies[autoen.dims[dim-1],
                     "Model Parameters"] = paste0("K = ", best.k,
                                                  ", Activation function = ",
```

```
                                        activ,". No. hidden layers = ",
                                        hidden.nodes)

}



#convert validation accuracies to dataframe
valid.accuracy = data.frame(valid.accuracy)

#column names
colnames(valid.accuracy) = c(
                          "Autoenc2",
                          "Autoenc3",
                          "Autoenc4",
                          "Autoenc5",
                          "Autoenc6")

#plot mean cv accuracies
ggplot(valid.accuracy,
       aes(x = 1:100)) +
  geom_line(aes(y = Autoenc2, color = "Autoenc2"), linetype = "solid") +
  geom_line(aes(y = Autoenc3, color = "Autoenc3"), linetype = "solid") +
  geom_line(aes(y = Autoenc4, color = "Autoenc4"), linetype = "solid") +
  geom_line(aes(y = Autoenc5, color = "Autoenc5"), linetype = "solid") +
  geom_line(aes(y = Autoenc6, color = "Autoenc6"), linetype = "solid") +
  labs(title = "",
       x = "Number of neighbours (K)",
       y = "Mean CV accuracy",
       color = "Legend")+
    theme_minimal() +
  scale_color_manual(values = c("darkblue", "darkred", "darkgreen", "darkorange", "black"),
                     labels = c("Autoenc2", "Autoenc3", "Autoenc4", "Autoenc5", "Autoenc6"))


#name of classes or folders
folders <- c("C", "G")

#function that reads in all images in a folder and returns a list of width x height image pixel data
read.images <- function(path.to.data, folder) {

  # get names of images in folder
  image.names = list.files(paste0(path.to.data,
                                  folder, "/"),
                           pattern = ".png",
```

```
                    full.names = TRUE)

  #empty list to store pixel data
  pixels.list = list()

  #for loop to get pixel data for each image
  for (image_name in image.names) {

    #Read in the image
    img <- readPNG(image_name,
                   native=FALSE) #output is an array not a native raster

    #append pixel image data to list
    pixels.list = append(pixels.list, list(img))


    # Convert the image to pixel values
   # pixels <- as.vector(img)
    #pixels.list = append(pixels.list, pixels)
  }

  #return pixel data of all images
   return(pixels.list)
}

#pixel data for class C
pixels.list.class.C = read.images("./data/", folders[1])

#pixel data for class G
pixels.list.class.G = read.images("./data/", folders[2])

##########################
#convert pixel data to dataframe

##C
datafr.pixels.class.C <- as.data.frame(matrix(unlist(pixels.list.class.C),
                                      ncol=28*28, byrow=T))

##G
datafr.pixels.class.G <- as.data.frame(matrix(unlist(pixels.list.class.G),
                                      ncol=28*28, byrow=T))

##########################

# Rename the columns
```

```
##C
colnames(datafr.pixels.class.C) <- paste0("pixel ",
                                  1:ncol(datafr.pixels.class.C))

##G
colnames(datafr.pixels.class.G) <- paste0("pixel ",
                                  1:ncol(datafr.pixels.class.G))

########################

#check dimensions of data
dim(datafr.pixels.class.C) #check first image of class C data
dim(datafr.pixels.class.G) #check first image of class G data

#########

#look at head of data: image 1 - first 5 rows and columns
head.C = round(head(datafr.pixels.class.C)[,1:5], 3) #C
head.G =  round(head(datafr.pixels.class.G)[,1:5], 3) #G

#########

#look at tail of data: image 1 - first 5  rows and columns
tail.C = round(tail(datafr.pixels.class.C)[, 780:784], 3) #C
tail.G = round(tail(datafr.pixels.class.G)[, 780:784], 3) #G

#########

#min and max values: sample image 1
min.C <- min(datafr.pixels.class.C) #C min
max.C <- max(datafr.pixels.class.C) #C max

min.G <- min(datafr.pixels.class.G) #C min
max.G <- max(datafr.pixels.class.G) #C max

#########

#look at data types
str(datafr.pixels.class.C)
str(datafr.pixels.class.G)
#########################################################################

# check for missing data
which(is.na(datafr.pixels.class.C) == TRUE) #C
```

```
which(is.na(datafr.pixels.class.G) == TRUE) #G

#add C class variable
datafr.pixels.class.C = datafr.pixels.class.C %>%
  mutate(class = rep("c", nrow(datafr.pixels.class.C)), .before="pixel 1")

#add image ID to C data
datafr.pixels.class.C = datafr.pixels.class.C %>%
  mutate(id = paste0("c", 1:nrow(datafr.pixels.class.C)), .before="pixel 1")

#add G class variable
datafr.pixels.class.G = datafr.pixels.class.G %>%
  mutate(class = rep("g", nrow(datafr.pixels.class.G)), .before="pixel 1")

#add image ID to G data
datafr.pixels.class.G = datafr.pixels.class.G %>%
  mutate(id = paste0("g", 1:nrow(datafr.pixels.class.G)), .before="pixel 1")

#combined CG dataset
data.cg = rbind(datafr.pixels.class.C, datafr.pixels.class.G)

#combined data dimensions
dim(data.cg)

#convert character and discrete variables to factors
data.cg$class = as.factor(data.cg$class)
data.cg$id = as.factor(data.cg$id)




#save image data
png("images/CG_hist.png")

par(mfrow=c(1, 2)) #set partition for plots

#histogram of C pixels
datafr.pixels.class.C %>% select(-class, -id)%>% #remove class and ID column
  as.matrix(ncol=28*28)%>%
  hist(col="darkseagreen2", main="A. Class C",
     xlab="Pixels")

#histogram of G pixels
datafr.pixels.class.G %>% select(-class, -id)%>% #remove class and ID column
  as.matrix(ncol=28*28)%>%
```

```
  hist(col="darkseagreen2", main="B. Class G",
     xlab="Pixels")

dev.off()

png("images/CG_visualization.png")
#Let us look at two images of C and G
par(mfrow=c(2, 2), mar = c(5, 4, 4, 8) + 0.1)


#NB:multiply by 255 to get pixel values 0 to 255
observation1 <- datafr.pixels.class.C[1, ] %>%
  select(-class, -id)%>%
  as.numeric()%>%
  matrix(nrow=28, byrow=F)


observation1867 <- datafr.pixels.class.C[1867, ] %>%
  select(-class, -id)%>%
  as.numeric()%>%
  matrix(nrow=28, byrow=F)


#rotate matrix 90 degrees in a clockwise direction
image1.class.C <- t(apply(observation1,
                          2, rev))

image1867.class.C <- t(apply(observation1867,
                          2, rev))


#plot image

image(image1.class.C, col=hcl.colors(2, "Grays"), asp = 1) #use grayscale
title("A. Observation 1")

image(image1867.class.C, col=hcl.colors(2, "Grays"), asp = 1) #use grayscale
title("B. Observation 1867")

##########################################################################

#NB:multiply by 255 to get pixel values 0 to 255
observation1 <- datafr.pixels.class.G[1, ] %>%
  select(-class, -id)%>%
  as.numeric()%>%
```

```
  matrix(nrow=28, byrow=F)


observation1866 <- datafr.pixels.class.G[1866, ] %>%
  select(-class, -id)%>%
  as.numeric()%>%
  matrix(nrow=28, byrow=F)


#rotate matrix 90 degrees in a clockwise direction
image1.class.G <- t(apply(observation1,
                          2, rev))

image1866.class.G <- t(apply(observation1866,
                          2, rev))


#plot image
image(image1.class.G,
      col=hcl.colors(2, "Grays"),
      asp = 1) #use grayscale
title("C. Observation 1")

image(image1866.class.G,
      col=hcl.colors(2, "Grays"),
      asp = 1) #use grayscale
title("D. Observation 1866")

dev.off()


#save image data
png("images/CG_barplot.png")

#calculate counts for each class of letters
C.counts <- datafr.pixels.class.C %>% nrow() #C
G.counts <- datafr.pixels.class.G %>% nrow() #G

#create dataframe of counts
CG.counts = data.frame(rbind(C.counts, G.counts))

#add column with letters variable
CG.counts = CG.counts %>% mutate(Letters = c("C", "G"))

#rename column name
```

```
colnames(CG.counts) = c("Counts", "Letters")

#remove rownames
rownames(CG.counts) = NULL

#create bar plot for letters counts
ggplot(CG.counts, aes(x=Letters, y=Counts)) +
  geom_bar(stat="identity", fill="steelblue") +
  geom_text(aes(label=Counts), vjust=-0.3) +
  theme_minimal()

dev.off()


#stratified sampling into 70% training and 30% test sets
set.seed(123)
cg.train.ind <- createDataPartition(data.cg$class, p = .7, list = FALSE, times = 1)

#70% training data
cg.train <- data.cg[cg.train.ind,]

#30% test data
cg.test <- data.cg[-cg.train.ind,]

#check for class imbalance
imbal.cg.train = table(cg.train$class)
imbal.cg.test = table(cg.test$class)

#randomly remove observation from training data C class
temp.train.c.data = cg.train %>% filter(class == "c") #filter train data only with class c

#sample one observation
set.seed(123)
rm.index = sample(1:nrow(temp.train.c.data), 1)

temp.train.c.data = temp.train.c.data[-rm.index, ] #remove index from temp data

temp.train.g.data = cg.train %>% filter(class == "g") #filter out g data

cg.train = temp.train.c.data %>% rbind(temp.train.g.data) #balanced cg data

#shuffle data
shuffled.ind = sample(nrow(cg.train))
cg.train = cg.train[shuffled.ind, ]
```

```
#check that data is balanced
bal.cg.train = table(cg.train$class)

#CG x-train data
cg.train.x = cg.train %>% select(-id, -class)

#CG y-train data
cg.train.y = cg.train %>% select(class)

#CG train IDs
cg.train.ids = cg.train %>% select(id)

##########################################
#CG x-test data
cg.test.x = cg.test %>% select(-id, -class)

#CG y-test data
cg.test.y = cg.test %>% select(class)

#CG test IDs
cg.test.ids = cg.test %>% select(id)



#load in saved fit results
knn.cg.original.fit = readRDS("images/knn.cg.original.fit.rds")

#10-fold cross validation
trControl <- trainControl(method  = "cv",
                          number  = 10,
                          savePredictions = "all")

#UNCOMMENT BELOW IF REQUIRED
#####################################################
#train KNN on original training data
#apply 10 fold cross validation
# set.seed(123)
# knn.cg.original.fit <- train(class ~ .,
#              method     = "knn",
#              tuneGrid   = expand.grid(k = 1:20),
#              trControl  = trControl,
#              metric     = "Accuracy",
#              data       = data.frame(class = cg.train.y,
#            cg.train.x))
```

```
#########################################################

#CV accuracy
cv.accuracy = knn.cg.original.fit

#best value of K
best.k = knn.cg.original.fit$bestTune$k


#save results
#saveRDS(knn.cg.original.fit, "images/knn.cg.original.fit.rds")
#plot CV accuracy
plot(knn.cg.original.fit,
        xlab = "Number of neighbours (k)",
     ylab = "Mean cross-validation accuracy")

#create dataframe with test accuracies
cg.test.accuracies = data.frame(matrix(ncol=3,
                                nrow = 26))

#assign column names
colnames(cg.test.accuracies) = c("Test Accuracy",
                                 "Model ID",
                                 "Model Parameters")

#add rownames as a column
rownames(cg.test.accuracies) = c("Original",
                                 "PCA2",
                                 "PCA3",
                                 "PCA4",
                                 "PCA5",
                                 "PCA6",
                                 "KPCA2.rbf",
                                 "KPCA3.rbf",
                                 "KPCA4.rbf",
                                 "KPCA5.rbf",
                                 "KPCA6.rbf",
                                 "KPCA2.poly",
                                 "KPCA3.poly",
                                 "KPCA4.poly",
                                 "KPCA5.poly",
                                 "KPCA6.poly",
                                 "Autoencoder2",
                                 "Autoencoder3",
                                 "Autoencoder4",
```

```
                                ”Autoencoder5”,
                                ”Autoencoder6”,
                                ”SOM2”,
                                ”SOM3”,
                                ”SOM4”,
                                ”SOM5”,
                                ”SOM6”)

#refit KNN model on train data using optimal value of K

set.seed(123)
best.cg.original.fit <- knn3(
                class ~ .,
                 data = data.frame(class = cg.train.y,
                                   cg.train.x),
                 k = best.k
                )

#model prediction on test set
predictions <- predict(best.cg.original.fit,
                       data.frame(cg.test.x),
                       type = ”class”)

#create dataframe of predictions vs observed
colnames(cg.test.y) = ”observed” #change column name
cg.true.vs.pred = cg.test.y %>% mutate(predictions)

#test confusion matrix
cg.test.metrics <- confusionMatrix(cg.true.vs.pred$predictions,
                                   cg.true.vs.pred$observed)

#assign test accuracy to dataframe
cg.test.accuracies[”Original”, ”Test Accuracy”] = round(cg.test.metrics$overall[”Accuracy”], 3)

#assign model ID
cg.test.accuracies[”Original”, ”Model ID”] = paste0(”cg.original.K”, best.k)

#assign model parameters
cg.test.accuracies[”Original”, ”Model Parameters”] = paste0(”K = ”,
                                   best.k)

#Extract confusion matrix
cg.original.conf.mat = cg.test.metrics$table
```

```
#apply pca using prcomp on scaled x data
pca.cg.fit = prcomp(cg.train.x)

#summary of pca analysis
summary.pca <- summary(pca.cg.fit)
#get eigenvalues or variance of each principal component
prcomp.variance <- pca.cg.fit$sdev^2

#put variance values in a dataframe
prcomp.variance = data.frame("PC"=1:784,
                             "Eigenvalues"= prcomp.variance)

############

# plot variance of principal component
p1 <- ggplot(prcomp.variance, aes( x = PC, y = Eigenvalues) ) +
  geom_line() +
  geom_hline( yintercept = 0.01, linetype = 'dotted', col = 'red') +
  theme_minimal() +
  xlab( 'Principal Component Number' ) +
  ylab( 'Eigenvalues' ) +
  ggtitle( 'A.' )

###############

#prcomp: proportion of variance
prcomp.variance$proportion.of.variance <- prcomp.variance$Eigenvalues/sum(prcomp.variance$Eigenvalues)

# plot proportion of variance of each principal component
p2 <- ggplot(prcomp.variance, aes( x = PC, y = proportion.of.variance) ) +
  geom_line() +
  geom_hline( yintercept = 0.01, linetype = 'dotted', col = 'red') +
  theme_minimal() +
  xlab( 'Principal Component Number' ) +
  ylab( 'Prop. Explained Var.' ) +
  ggtitle( 'B.' )

###############

#prcomp: cumulative proportion of variance
prcomp.variance$cum.variance <- cumsum(prcomp.variance$proportion.of.variance)

# plot cumulative variance
p3 <- ggplot(prcomp.variance, aes( x = PC, y = cum.variance) ) +
  geom_line() +
```

```
  geom_hline( yintercept = 0.95, linetype = 'dotted', col = 'red') +
  theme_minimal() +
  xlab( 'Principal Component Number' ) +
  ylab( 'Cum. Prop. Explained Var.' ) +
  ggtitle( 'C.' )

#grid plot
grid.arrange( p1, p2, p3,
              ncol=2)

##############################


#read in data to reduce run time
cg.test.accuracies = readRDS("images/cg.test.pca.accuracies.rds")
valid.accuracy = readRDS("images/cg.valid.pca.accuracy.rds")

#UNCOMMENT BELOW IF REQUIRED

####################
# #dimensions to assess
# dimensions = c(2, 3, 4, 5, 6)
#
# #rownames
# PCA.dims = c("PCA2", "PCA3", "PCA4", "PCA5", "PCA6")
#
# #validation accuracies
# valid.accuracy = c()
#
# ####################
#
# for (dim in dimensions) {
#
#   #train scores
#   pca.cg.train.scores = pca.cg.fit$x[,1:dim]
#
#   ####################
#
#   #test scores
#   pca.cg.test.scores <- data.frame(predict(pca.cg.fit,
#                            newdata = cg.test.x))
#
#   #choose dimensions for test set
#   pca.cg.test.scores = pca.cg.test.scores[,1:dim]
#
```

```
#   ####################
#   #Find optimal value of K
#
#   #10-fold cross validation
#   trControl <- trainControl(method  = "cv",
#                             number  = 10,
#                             savePredictions = "all")
#
#   #apply 10 fold cross validation
#   set.seed(123)
#   knn.cg.pca.fit <- train(class ~ .,
#               method     = "knn",
#               tuneGrid   = expand.grid(k = 1:20),
#               trControl  = trControl,
#               metric     = "Accuracy",
#               data       = data.frame(class = cg.train.y,
#             pca.cg.train.scores))
#
#   #CV accuracy
#   cv.accuracy = knn.cg.pca.fit
#
#   #best value of K
#   best.k = knn.cg.pca.fit$bestTune$k
#
#   #validation accuracies
#   valid.accuracy = cbind(valid.accuracy,
#                          cv.accuracy$results$Accuracy)
#
#
#   ###################
#   #refit KNN model on train data using optimal value of K
#   set.seed(123)
#   best.cg.pca.fit <- knn3(
#                   class ~ .,
#                   data = data.frame(class = cg.train.y,
#                                     pca.cg.train.scores),
#                   k = best.k
#                  )
#
#   ####################
#   #model prediction on test set
#   predictions <- predict(best.cg.pca.fit,
#                      data.frame(pca.cg.test.scores),
#                      type = "class")
#
```

```
#   #create dataframe of predictions vs observed
#   colnames(cg.test.y) = "observed" #change column name
#   cg.true.vs.pred = cg.test.y %>% mutate(predictions)
#
#   #test confusion matrix
#   cg.test.metrics <- confusionMatrix(cg.true.vs.pred$predictions,
#                                      cg.true.vs.pred$observed)
#
#   ######################
#   #assign test accuracy to dataframe
#   cg.test.accuracies[PCA.dims[dim-1], "Test Accuracy"] = round(cg.test.metrics$overall["Accuracy"], 3)
#
#   #assign model ID
#   cg.test.accuracies[PCA.dims[dim-1], "Model ID"] = paste0(paste0("cg.pca.K", best.k, ".D", dim))
#
#
#   #assign model parameters
#   cg.test.accuracies[PCA.dims[dim-1], "Model Parameters"] = paste0("K = ",
#                                    best.k)
#
# }


#########################################
#UNCOMMENT CODE BELOW IF REQUIRED TO SAVE ABOVE RESULTS

#save rds data
# saveRDS(cg.test.accuracies, "images/cg.test.pca.accuracies.rds")
# saveRDS(valid.accuracy, "images/cg.valid.pca.accuracy.rds")



#convert validation accuracies to dataframe
valid.accuracy = data.frame(valid.accuracy)

#column names
colnames(valid.accuracy) = PCA.dims

#plot mean cv accuracies
ggplot(valid.accuracy,
       aes(x = 1:20)) +
  geom_line(aes(y = PCA2, color = "PCA2"), linetype = "solid") +
  geom_line(aes(y = PCA3, color = "PCA3"), linetype = "solid") +
  geom_line(aes(y = PCA4, color = "PCA4"), linetype = "solid") +
  geom_line(aes(y = PCA5, color = "PCA5"), linetype = "solid") +
```

```
  geom_line(aes(y = PCA6, color = "PCA6"), linetype = "solid") +
  labs(title = "",
       x = "Number of neighbours (k)",
       y = "Mean CV accuracy",
       color = "Legend")+
    theme_minimal() +
  scale_color_manual(values = c("darkblue", "darkred", "darkgreen", "darkorange", "black"),
                     labels = c("PCA2", "PCA3", "PCA4", "PCA5", "PCA6"))



#read in data
cg.test.accuracies = readRDS("images/cg.test.kpca.accuracies.rds")

#dimensions to assess
dimensions = c(2, 3, 4, 5, 6)

#Kernel functions
Kernels = c("rbfdot","polydot")

#rownames
KPCA.dims = c("KPCA2.rbf", "KPCA3.rbf", "KPCA4.rbf", "KPCA5.rbf", "KPCA6.rbf",
              "KPCA2.poly", "KPCA3.poly", "KPCA4.poly", "KPCA5.poly", "KPCA6.poly")

#validation accuracies
valid.accuracy = c()

# Define training control
train.control <- trainControl(method = "cv", number = 10)

#UNCOMMENT BELOW IF NECESSARY
####################

# for (dim in dimensions) {
#   for (kern in Kernels) {
#
#
#     if (kern == "rbfdot") {
#       set.seed(123)
#           kpca.cg.fit <- kpca(~.,
#                 data = data.frame(cg.train.x),
#                 kernel = kern,
#                 kpar = list(sigma = 0.1),
#                 features = dim)
```

```
#
#   } else { #polydot
#       set.seed(123)
#       kpca.cg.fit <- kpca(~.,
#               data = data.frame(cg.train.x),
#               kernel = kern,
#               kpar = list(degree = 1,
#                           scale = 1, offset = 1),
#               features = dim)
#   }
#
#    #train scores
#    kpca.cg.train.scores = kpca.cg.fit@pcv
#
#    ####################
#
#    #test scores
#    kpca.cg.test.scores <- predict(kpca.cg.fit,
#                            data.frame(cg.test.x))
#    ####################
#
#    #Find optimal value of K
#
#    #10-fold cross validation
#    trControl <- trainControl(method  = "cv",
#                           number  = 10,
#                           savePredictions = "all")
#
#    #apply 10 fold cross validation
#    set.seed(123)
#    knn.cg.kpca.fit <- train(class ~ .,
#                method     = "knn",
#                tuneGrid   = expand.grid(k = 1:20),
#                trControl  = trControl,
#                metric     = "Accuracy",
#                data       = data.frame(class = cg.train.y,
#               kpca.cg.train.scores))
#
#    #CV accuracy
#    cv.accuracy = knn.cg.kpca.fit
#
#    #best value of K
#    best.k = knn.cg.kpca.fit$bestTune$k
#
#    #validation accuracy
```

```
#     valid.accuracy = cbind(valid.accuracy,
#                            cv.accuracy$results$Accuracy)
#
#     ###################
#     #refit KNN model on train data using optimal value of K
#     set.seed(123)
#     best.cg.kpca.fit <- knn3(
#                     class ~ .,
#                     data = data.frame(class = cg.train.y,
#                                       kpca.cg.train.scores),
#                     k = best.k
#                    )
#
#     ####################
#
#     #model prediction on test set
#     predictions <- predict(best.cg.kpca.fit,
#                        data.frame(kpca.cg.test.scores),
#                        type = "class")
#
#     #create dataframe of predictions vs observed
#     colnames(cg.test.y) = "observed" #change column name
#     cg.true.vs.pred = cg.test.y %>% mutate(predictions)
#
#     #test confusion matrix
#     cg.test.metrics <- confusionMatrix(cg.true.vs.pred$predictions,
#                                        cg.true.vs.pred$observed)
#
#     ######################
#
#     #offset for each Kernel function - used to populate test accuracy table
#     if (kern == Kernels[1]) {
#       offset = 0
#       sigma = 0.1
#       degree = NA #HERE
#       scale = NA #here
#       offset.model.param = NA #here
#     } else if (kern == Kernels[2]) {
#       offset = +5
#       sigma = NA
#       degree = 1
#       scale = 1 #here
#       offset.model.param = 1 #here
#
#     }
```

```
#
#     #assign test accuracy to dataframe
#     cg.test.accuracies[KPCA.dims[dim-1+offset],
#                        "Test Accuracy"] = round(cg.test.metrics$overall["Accuracy"], 3)
#
#     #assign model ID
#     cg.test.accuracies[KPCA.dims[dim-1+offset], #-1 because dims start at 2
#                        "Model ID"] = paste0("cg.kpca.K",
#                                             best.k, ".",
#                                             kern,
#                                             ".D",
#                                             dim)
#
#     #assign model parameters
#     cg.test.accuracies[KPCA.dims[dim-1+offset],
#                        "Model Parameters"] = paste0("K = ",
#                                    best.k,
#                                    ", Kernel = ",
#                                    kern,
#
#                                    ifelse(!is.na(sigma),
#                                           paste0(", sigma = ",
#                                           sigma), ""),
#
#                                    ifelse(!is.na(degree), #here
#                                           paste0(", degree = ",
#                                           degree), ""),
#
#
#                                     ifelse(!is.na(offset), #here
#                                           paste0(", offset = ",
#                                           offset.model.param), ""),
#
#
#                                    ifelse(!is.na(scale), #here
#                                           paste0(", scale = ",
#                                           scale), "")
#                                    )
#   }
# }

#########################################
#UNCOMMENT CODE BELOW IF REQUIRED TO SAVE ABOVE RESULTS

#save rds data
```

```
# saveRDS(cg.test.accuracies, "images/cg.test.kpca.accuracies.rds")


#read in data
cg.test.accuracies = readRDS("images/cg.test.som.accuracies.rds")

#################################
#ensure data is a matric
check = is.matrix(cg.train.x)

#convert cg train x data to matrix format
check = cg.train.x = as.matrix(cg.train.x)

#ensure data is numeric
check = is.numeric(cg.test.x)

#convert cg train x data to matrix format
cg.test.x = as.matrix(cg.test.x)

#####################################

#dimensions to assess
dimensions = c(2, 3, 4, 5, 6)

#rownames
SOM.dims = c("SOM2", "SOM3", "SOM4", "SOM5", "SOM6")

#validation accuracies
valid.accuracy = c()

# Define training control
train.control <- trainControl(method = "cv", number = 10)

#UNCOMMENT BELOW IF NECESSARY
#####################################

# #for loop to:
#  ##train SOM model on training data, where SOM model:
#   ### is trained on training data
#   ###a given dim x dim dimensions
#
#  ##Make predictions on test and training set to get reduced features
#  ##build knn model on reduced training data and apply 10 fold CV
#  ## determine optimal K with largest CV accuracy
#  ##rebuild knn model on reduced train data using optimal K
```

```
#  ##predict test data label using reduced test data and knn model
#  ##determine test data accuracy by assessing confusion matrix
#
# #topologies
# topologies = c("hexagonal", "rectangular")
#
# #best cv accuracy given the best k
# ##results given per topology
# best.k.cv.accuracy = data.frame(hexagonal = double(),
#                                 rectangular = double())
#
# #for each toplogy and dimension combination...
# for (topo in topologies) {
#   for (dim in dimensions) {
#
#     #SOM grid
#     som.grid <- kohonen::somgrid(xdim = dim,
#                          ydim=dim,
#                          topo=topo) #hexagonal or rectangular
#
#     #build SOM model
#     set.seed(123)
#     som.model <- kohonen::som(cg.train.x,
#                       grid=som.grid,
#                       rlen=500, #number of iterations
#                       alpha=c(0.05,0.01), #learning rate
#                       keep.data = TRUE
#                       )
#
#     #predict train reduced data
#     set.seed(123)
#     som.reduced.train = predict(som.model, cg.train.x)$predictions[[1]]
#
#     #10-fold cross validation
#     trControl <- trainControl(method  = "cv",
#                               number  = 10,
#                               savePredictions = "all")
#
#     #Apply KNN on SOM model with given dimension and topology:
#     #apply 10 fold cross validation
#     set.seed(123)
#     knn.cg.som.fit <- train(class ~ .,
#                         method     = "knn",
#                         tuneGrid   = expand.grid(k = 1:20), #K grid search
#                         trControl  = trControl,
```

```
#                       metric     = "Accuracy",
#                       data       = data.frame(class = cg.train.y,
#                     som.reduced.train)) #take in SOM reduced train data
#
#     #obtain CV accuracies for each value of k
#     cv.accuracy = knn.cg.som.fit
#
#     #best value of K with largest cv accuracy
#     best.k = knn.cg.som.fit$bestTune$k
#
#     #extract largest valid accuracy given best k
#     #append to dataframe
#     best.k.cv.accuracy[dim-1, topo] = cv.accuracy$results$Accuracy[best.k]
#   }
# }
#
# ###################################
#
# #for each dimension 2 to 6
# for (dim in dimensions) {
#
#   #Find best topology with greated CV accuracy
#   #topology = hexagonal if CV accuracy for hexagonal>= rectangular
#   if (best.k.cv.accuracy[dim-1, "hexagonal"] >= best.k.cv.accuracy[dim-1, "rectangular"]) { topo = "hexagonal"
#   } else {
#     topo = "rectangular"
#   }
#
#   #SOM grid
#   som.grid <- kohonen::somgrid(xdim = dim,
#                         ydim=dim,
#                         topo=topo) #best topology
#
#   #build SOM model
#   set.seed(123)
#   som.model <- kohonen::som(cg.train.x,
#                       grid=som.grid,
#                       rlen=500,
#                       alpha=c(0.05,0.01),
#                       keep.data = TRUE
#                       )
#
#   #predict train reduced data
#   set.seed(123)
#   som.reduced.train = predict(som.model, cg.train.x)$predictions[[1]]
```

```
#
#   #predict test reduced data
#   set.seed(123)
#   som.reduced.test = predict(som.model, cg.test.x)$predictions[[1]]
#
#   #Find optimal value of K
#
#   #10-fold cross validation
#   trControl <- trainControl(method  = "cv",
#                             number  = 10,
#                             savePredictions = "all")
#
#   #Apply KNN on SOM model with given dimension and topology:
#   #apply 10 fold cross validation
#   set.seed(123)
#   knn.cg.som.fit <- train(class ~ .,
#                     method     = "knn",
#                     tuneGrid   = expand.grid(k = 1:20), #K grid search
#                     trControl  = trControl,
#                     metric     = "Accuracy",
#                     data       = data.frame(class = cg.train.y,
#                    som.reduced.train)) #take in SOM reduced train data
#
#   #obtain CV accuracies for each value of k
#   cv.accuracy = knn.cg.som.fit
#
#   #best value of K with largest cv accuracy
#   best.k = knn.cg.som.fit$bestTune$k
#
#   #validation accuracy for each value of k
#   valid.accuracy = cbind(valid.accuracy,
#                          cv.accuracy$results$Accuracy)
#
#
#   ###################
#   #refit KNN model on train data using optimal value of K
#   set.seed(123)
#   best.cg.som.fit <- knn3(
#                     class ~ .,
#                     data = data.frame(class = cg.train.y,
#                                       som.reduced.train),
#                     k = best.k
#                     )
#
#   ####################
```

```
#
#   #model prediction on test set
#   predictions <- predict(best.cg.som.fit,
#                          data.frame(som.reduced.test),
#                          type = "class")
#
#   #create dataframe of predictions vs observed
#   colnames(cg.test.y) = "observed" #change column name
#     cg.true.vs.pred = cg.test.y %>% mutate(predictions)
#
#   #test confusion matrix
#   cg.test.metrics <- confusionMatrix(cg.true.vs.pred$predictions,
#                                      cg.true.vs.pred$observed)
#
#    ##########################
#
#   #assign test accuracy to dataframe
#   cg.test.accuracies[SOM.dims[dim-1], #-1 because dims start at 2
#                          "Test Accuracy"] = round(cg.test.metrics$overall["Accuracy"], 3)
#
#   #assign model ID
#   cg.test.accuracies[SOM.dims[dim-1], #-1 because dims start at 2
#                          "Model ID"] = paste0("cg.som.K",
#                                               best.k,
#                                               ".", substr(topo, 1, 3),
#                                               ".D",
#                                               dim)
#
#   #assign model parameters
#   cg.test.accuracies[SOM.dims[dim-1],
#                          "Model Parameters"] = paste0("K = ", best.k,
#                                                       ", topology = ", topo,
#                                                       ", rlen = 500",
#                                                       "alpha = [0.05, 0.01]")
#
# }


#########################################
#UNCOMMENT CODE BELOW IF REQUIRED TO SAVE ABOVE RESULTS

#save rds data
#saveRDS(cg.test.accuracies, "images/cg.test.som.accuracies.rds")
```

```
#read in results
cg.autoen.valid.results = readRDS("images/cg.autoen.valid.results.rds")
cg.test.autoen.accuracies =  readRDS("images/cg.test.autoen.accuracies.rds")

#UNCOMMENT CODE BELOW IF REQUIRED.
#SET UP CONDA ENVIRONMENT!
# ##################################################################

#Set random seed
# set.seed(123)
# tensorflow::set_random_seed(123)
#
#
# #GRIDSEARCH
# ##Ensure butterfly shape
# input.nodes = dim(cg.train.x)[2] #also hidden nodes
#
# hidden.nodes = c(500, 300, 100) #less than input nodes but greater than dimensions
#
# #dimensions
# dimensions = c(2, 3, 4, 5, 6) #bottleneck
#
# #activation functions
# activation.grid = c("relu", "tanh")
#
# #Make the target variable is numeric, starting at 0
# cg.train.y.auto <- as.integer(factor(cg.train.y$class)) - 1 #train
# cg.test.y.auto <- as.integer(factor(cg.test.y$observed)) - 1 #test
#
# #save a copy of original y-data before hot-coding
# cg.train.y.auto.original <- cg.train.y.auto #train
# cg.test.y.auto.original <- cg.test.y.auto #test
#
# #hot-coding
# cg.train.y.auto <- to_categorical(cg.train.y.auto) #train
# cg.test.y.auto <- to_categorical(cg.test.y.auto) #test
#
# #ensure numeric, matrix feature data
# is.matrix(cg.train.x) #train
# is.matrix(cg.test.x) #test

##################################################################
#DEFINE NUMBER OF FOLDS
# num.folds <- 10
#
```

```
# #Split the data into k-folds
# set.seed(123)
# folds <- createFolds(cg.train.y$class, k = num.folds)
#
# #empty dataframe for validation results
# cg.autoen.valid.results = data.frame(valid.accuracy = c(),
#                                      parameters = c())

#################################################################
#PERFORM GRIDSEARCH
# for (hidden.node in hidden.nodes) {
#   for (dim in dimensions) {
#     for (activ in activation.grid) {
#       ###################################
#       #empty vectors for loss at each fold...
#       #accuracy
#       valid.loss.values <- numeric(num.folds) #validation
#
#       ###########################################
#
#       #for each fold...
#       for (fold in 1:num.folds) {
#
#       #extract train and validation folds
#       train.indices <- unname(unlist(folds[-fold]))
#       valid.indices <- unname(unlist(folds[fold]))
#
#       #prepare features at each fold
#       x.train <- cg.train.x[train.indices, ] #train
#       x.valid <- cg.train.x[valid.indices, ] #validation
#
#       #prepare target data at each fold
#       y.train <- cg.train.x[train.indices,] #train
#       y.valid <- cg.train.x[valid.indices,] #validation
#
#       ###########################################
#
#        #fit the model on the training-fold data
#        #with parameters at the current iteration
#
#        #initialize model
#        set.seed(123)
#        autoen.model <- keras_model_sequential()
#
#        #define the model
```

```
#         autoen.model%>%
#         layer_dense(units = hidden.node,
#                     activation = activ,
#                     input_shape = dim(x.train)[2]) %>%
#         layer_dense(units = dim, activation = activ, name = "bottleneck")%>%
#         layer_dense(units = hidden.node, activation = activ) %>%
#         layer_dense(units = dim(x.train)[2],
#                     activation = "sigmoid")
#
#         ##compile model
#         autoen.model %>% compile(
#           loss = "mse",
#           optimizer = 'adam',
#          )
#
#         ##train the model on train-fold data
#         set.seed(123)
#         autoen.model %>% fit(
#           x = x.train,
#           y = x.train,
#           epochs = 25,
#           batch_size=128,
#           verbose = 0)
#       ##############################################
#
#       #Predictions on valid data
#        set.seed(123)
#        valid.metrics <- autoen.model %>% evaluate(x.valid, x.valid)
#
#        ##valid loss
#        valid.loss <- unname(valid.metrics[1])
#
#        #append validation loss for this fold
#         valid.loss.values = c(valid.loss.values,
#                               valid.loss)
#       } #end of 10 folds
#
#       ##############################################
#
#       # Calculate the mean results across all folds
#
#       ##LOSS
#       mean.valid.loss <- mean(valid.loss.values) #validation
#
#      ##############################################
```

```
#
#       # Store the results
#       cg.autoen.valid.results = rbind(cg.autoen.valid.results,
#                                       c(mean.valid.loss,
#                                         paste0("No. of hidden nodes = ",
#                                                hidden.node,
#                                                ". Dimensions = ",
#                                                dim,
#                                                ". Activation function = ",
#                                                activ)))
#     }
#   }
# }

#########################################
#save results
#saveRDS(cg.autoen.valid.results, "images/cg.autoen.valid.results.rds")

#########################################

#change column names
# colnames(cg.autoen.valid.results) = c("Mean CV Loss", "Parameters")
#
# #round off loss to 3 decimal places
# cg.autoen.valid.results$`Mean CV Loss` = round(as.double(cg.autoen.valid.results$`Mean CV Loss`), 3)
#
# #separate hyperparameters into separate columns
# cg.autoen.valid.results <- separate(cg.autoen.valid.results,
#                                     "Parameters",
#                                     into = c("No. of hidden nodes",
#                                              "Dimensions",
#                                              "Activation function"),
#                                     sep = "\\. Dimensions = |\\. Activation function = ")
#
# #extract number of hidden nodes
# cg.autoen.valid.results$`No. of hidden nodes` = gsub(".*=\\s*", "",
#      cg.autoen.valid.results$`No. of hidden nodes`)
#
#
# #ensure numeric values
# ##hidden nodes
# cg.autoen.valid.results$`No. of hidden nodes` = as.integer(cg.autoen.valid.results$`No. of hidden nodes`)
#
# ##dimensions
# cg.autoen.valid.results$Dimensions = as.integer(cg.autoen.valid.results$Dimensions)
```

```
#
# ########################################
#
# #RETRAIN AUTOENCODER WITH OPTIMAL HYPERPARAMETERS
#
# #empty validation vector
# valid.accuracy = c()
#
# #AUTOENCODER DIMENSION IDs
# autoen.dims = c("Autoencoder2",
#                 "Autoencoder3",
#                 "Autoencoder4",
#                 "Autoencoder5",
#                 "Autoencoder6")
#
#
#
# for (dim in dimensions) {
#
#   #extract log loss values for specific dimension
#   optimal.logloss = cg.autoen.valid.results%>%
#     filter(Dimensions == dim)
#
#   #find smallest CV reconstruction error
#   min.row = which.min(apply(optimal.logloss,
#                             1,
#                             function(x) x[1]))
#
#   #get parameters for min CV Loss
#   hidden.nodes = optimal.logloss[min.row, "No. of hidden nodes"]
#
#   activ = optimal.logloss[min.row, "Activation function"]
#
#   ################################
#
#   #initialize model
#   set.seed(123)
#   tensorflow::set_random_seed(123)
#   autoen.model <- keras_model_sequential()
#
#   #define the model
#   autoen.model%>%
#     layer_dense(units = hidden.nodes,
#                 activation = activ,
#                 input_shape = dim(cg.train.x)[2]) %>%
```

```
#     layer_dense(units = dim,
#                 activation = activ,
#                 name = "bottleneck")%>%
#             layer_dense(units = hidden.nodes,
#                         activation = activ) %>%
#             layer_dense(units = dim(cg.train.x)[2],
#                         activation = "sigmoid")
#
#   ##compile model
#   autoen.model %>% compile(
#             loss = "mse",
#             optimizer = 'adam',
#           )
#
#   ##train the model on train-fold data
#   set.seed(123)
#   autoen.model %>% fit(
#   x = cg.train.x,
#   y = cg.train.x,
#   epochs = 25,
#   batch_size=128,
#   verbose = 0)
#
#   ##########################################
#   #Reduced train data
#    autoen.reduced.train = predict(autoen.model, cg.train.x)
#
#   #column names for train data
#   colnames(autoen.reduced.train) = colnames(cg.train.x)
#
#   ##################################################
#
#   #Reduced test data
#   autoen.reduced.test = predict(autoen.model, cg.test.x)
#
#   #column names for test data
#   colnames(autoen.reduced.test) = colnames(cg.test.x)
#
#   ##################################################
#
#   #BUILD KNN MODEL
#
#   #10-fold cross validation
#   trControl <- trainControl(method  = "cv",
#                             number  = 10,
```

```
#                                savePredictions = "all")
#
#  #Apply KNN on SOM model with given dimension and topology:
#  #apply 10 fold cross validation
#  set.seed(123)
#  knn.cg.autoen.fit <- caret::train(class ~ .,
#                      method     = "knn",
#                      tuneGrid   = expand.grid(k = 1:20), #K grid search
#                      trControl  = trControl,
#                      metric     = "Accuracy",
#                      data       = data.frame(class = cg.train.y,
#                    autoen.reduced.train)) #autoencoder reduced train data
#
#  #obtain CV accuracies for each value of k
#  cv.accuracy = knn.cg.autoen.fit
#
#  #best value of K with largest cv accuracy
#  best.k = knn.cg.autoen.fit$bestTune$k
#
#  #validation accuracy for each value of k
#  valid.accuracy = cbind(valid.accuracy,
#                         cv.accuracy$results$Accuracy)
#
#  ###################
#  #refit KNN model on train data using optimal value of K
#  set.seed(123)
#  best.cg.autoen.fit <- knn3(
#                      class ~ .,
#                      data = data.frame(class = cg.train.y,
#                                        autoen.reduced.train),
#                      k = best.k
#                     )
#  ####################
#  #model prediction on test set
#  predictions <- predict(best.cg.autoen.fit,
#                         data.frame(autoen.reduced.test),
#                         type = "class")
#
#  #create dataframe of predictions vs observed
#  colnames(cg.test.y) = "observed" #change column name
#  cg.true.vs.pred = cg.test.y %>% mutate(predictions)
#
#  #test confusion matrix
#  cg.test.metrics <- confusionMatrix(cg.true.vs.pred$predictions,
#                                     cg.true.vs.pred$observed)
```

```
#
#   #assign test accuracy to dataframe
#   cg.test.accuracies[autoen.dims[dim-1], #-1 because dims start at 2
#                      "Test Accuracy"] = round(cg.test.metrics$overall["Accuracy"], 3)
#
#   #assign model ID
#   cg.test.accuracies[autoen.dims[dim-1], #-1 because dims start at 2
#                      "Model ID"] = paste0("cg.autoen.K",
#                                           best.k,
#                                           ".D",
#                                           dim)
#
#   #assign model parameters
#   cg.test.accuracies[autoen.dims[dim-1],
#                      "Model Parameters"] = paste0("K = ", best.k,
#                                                   ", Activation function = ",
#                                                   activ,". No. hidden layers = ",
#                                                   hidden.nodes)
#
# }

#########################################
#UNCOMMENT CODE BELOW IF REQUIRED TO SAVE ABOVE RESULTS

#save rds data
#saveRDS(cg.test.accuracies, "images/cg.test.autoen.accuracies.rds")

#####################################
#get absolute difference between each reduction technique and the original dataset

## WDBC
wdbc.test.accuracies$`Difference` <- abs(wdbc.test.accuracies$`Test Accuracy` - wdbc.test.accuracies$`Test Accuracy`[1])

## CG
cg.test.autoen.accuracies$`Difference` <- abs(cg.test.autoen.accuracies$`Test Accuracy` - cg.test.autoen.accuracies$`Test Ac

#########################################
#Bind model accuracies with empty row
model.comparison = rbind(wdbc.test.accuracies,
    cg.test.autoen.accuracies, row.names=TRUE)


##############################
#SOME EDITS
```

```
#change number of hidden layers to number of nodes in hidden layers
model.comparison$`Model Parameters` = sub("No. hidden layers",
    "No. of nodes in hidden layers",
    model.comparison$`Model Parameters`)

#convert 500alpha to 500, alpha
model.comparison$`Model Parameters` = sub("500alpha",
    "500, alpha",
    model.comparison$`Model Parameters`)

#remove row
model.comparison = model.comparison[-c(nrow(model.comparison)), ]

model.comparison$`Model ID` <- sub("rbfdot", "rbf", model.comparison$`Model ID`)

model.comparison$`Model ID` <- sub("polydot", "poly", model.comparison$`Model ID`)
############################

#remove row namess
rownames(model.comparison) = NULL

model.comparison = model.comparison %>% select( `Model ID`, `Model Parameters` , `Test Accuracy`, `Difference`)
####################

kable(model.comparison,
      caption="Model Performance and Comparison of KNN Trained on Reduced WDBC and CG Data Obtained by Various
      format="markdown",
          booktabs = TRUE,
      longtable=T)%>%
  add_footnote("Model ID:", notation="none")%>%
  add_footnote(c("Data source: cg or wdbc",
                 "Data type: original, pca, kpca, autoen or som",
                 "No. neighbours: K#",
                 "Kernel: rbf or poly",
                 "Topology: hex or rec",
                 "Dimension: D#"),
               notation = "number")


#variable names
variables = colnames(original.data.wdbc)

#variable types
variable.type = c("ID", "Dependent", rep("Independent", 30))
```

```
#data types
data.type = c("Categorical", "Categrical",
              rep("Continuous", 30))
#data.description =

#create tibble with data types
datatypes.wdbc = tibble(`Variable name` = variables,
                        `Variable type` = variable.type,
                        `Data type` = data.type)
#print table
kable(datatypes.wdbc,
      caption="WDBC variable- and data- types",
      format="markdown")
kable(head.wdbc[,1:8],
      caption="Head of WDBC dataset, with columns 1 to 8",
      format="markdown")


kable(head.wdbc[,9:16],
      caption="Head of WDBC dataset, with columns 8 to 16",
      format="markdown")


kable(head.wdbc
      [,17:24],
      caption="Head of WDBC dataset, with columns 17 to 24",
      format="markdown")


kable(head.wdbc[,25:32],
      caption="Head of WDBC dataset, with columns 25 to 32",
      format="markdown")

kable(tail.wdbc[,1:8],
      caption="Tail of WDBC dataset, with columns 1 to 8",
      format="markdown")

kable(tail.wdbc[,9:16],
      caption="Tail of WDBC dataset, with columns 8 to 16",
      format="markdown")

kable(tail.wdbc[,17:24],
      caption="Tail of WDBC dataset, with columns 17 to 24",
      format="markdown")
```

```
kable(tail.wdbc[,25:32],
      caption="Tail of WDBC dataset, with columns 25 to 32",
      format="markdown")
#read in RDS
plot_list = readRDS("images/hist_of_WDBC_mean_variables.rds")

# Arrange all histograms in the same grid
grid.arrange(grobs = plot_list, ncol = 3)

#read in RDS
plot_list = readRDS("images/hist_of_WDBC_se_variables.rds")

# Arrange all histograms in the same grid
grid.arrange(grobs = plot_list, ncol = 3)
#read in RDS
plot_list = readRDS("images/hist_of_WDBC_worst_variables.rds")

# Arrange all histograms in the same grid
grid.arrange(grobs = plot_list, ncol = 3)

#read in RDS data
counts = readRDS("images/barplot_of_WDBC_class.rds")

#create bar plot for diagnosis variable
ggplot(counts, aes(x=diagnosis, y=count)) +
  geom_bar(stat="identity", fill="steelblue") +
  geom_text(aes(label=count), vjust=-0.3) +
  theme_minimal()


#create confusion matrix table
kable(wdbc.original.conf.mat,
      caption="Confusion matrix showing predictions made on the original, high-dimensional WDBC test data, where the row
      format="markdown")

kable(head.C,
      caption="Head of the CG dataset for class C, with pixel values 1 to 5 (columns) and images 1 to 6 (rows)",
      format="markdown")

kable(head.G,
      caption="Head of the CG dataset for class G, with pixel values 1 to 5 (columns) and images 1 to 6 (rows)",
      format="markdown")
```

```
kable(tail.C,
      caption="Tail of the last 6 images (rows) in the CG dataset for class C, with pixel values 780 to 784 (columns)",
      format="markdown")

kable(tail.G,
      caption="TTail of the last 6 images (rows) in the CG dataset for class G, with pixel values 780 to 784 (columns)",
      format="markdown")

#load image data
img = readPNG("images/CG_hist.png")
grid.raster(img)

#load image data
img = readPNG("images/CG_visualization.png")
grid.raster(img)


#save image data
img = readPNG("images/CG_barplot.png")
grid.raster(img)

kable(cg.original.conf.mat,
      caption="Confusion matrix showing predictions made on the original, high-dimensional CG test data, where the rows re
      format="markdown")
```

11 References

Built In. 2022. "What Is the Curse of Dimensionality?" online. https://builtin.com/data-science/curse-dimensionality.

GeeksforGeeks. 2020. "Self Organizing Maps Kohonen Maps."

Karanam, Shashmi. 2021. "Curse of Dimensionality — a 'Curse' to Machine Learning." /url%7Bhttps://towardsdatascience.com/curse-of-dimensionality-a-curse-to-machine-learning-c122ee33bfeb%7D.

Khodabakhsh, Hojjat. 2023. "MNIST Dataset." https://www.kaggle.com/datasets/hojjatk/mnist-dataset.

Tanner, G. 2022. "Kernel PCA." https://ml-explained.com/blog/kernel-pca-explained.

UCI.edu. 2019. "UCI Machine Learning Repository." https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic.

Wikipedia. 2023a. "Autoencoder."

———. 2023b. "Principal Component Analysis." https://en.wikipedia.org/wiki/Principal_component_analysis.

———. 2023c. "Self-Organizing Map."