TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG

Die Ressourcenuniversität. Seit 1765.

Personal Programming Project

Winter Semester 2021/22

Machine learning to predict fatigue strength of steel from composition and processing parameters

by

Jerald Reventh Irudayaraj
Matriculation Number: 66191
Jerald-Reventh.Irudayaraj@student.tu-freiberg.de

Course supervisors:
Dr. Arun Prakash, Dr. Stefan Prüger

Institute of Mechanics and Fluid Dynamics(IMFD)
Master course of study: Computational Materials Science(CMS)
TU Bergakademie Freiberg

April 5, 2022

Contents

List of Abbreviations

PCA ..... Principal Component Analysis

K-means ..... K-means Clustering

CMatrix ..... Correlation Matrix

ANN ..... Artificial Neural Network

MLR ..... Multi Linear Regression

kNN ..... k-Nearest Neighbors

Std ..... Standard deviation

PC ..... Principal Components

GD ..... Gradient Descent

SGD ..... Stochastic Gradient Descent

SGDM ..... Stochastic Gradient Descent with Momentum

RMSP ..... Root Mean Square Propagation

Adam ..... Adaptive Momentum

MSE ..... Mean Squared Error Loss

RMSE ..... Root Mean Squared Error Loss

FP ..... Forward Propagation

BP ..... Backward Propagation

ReLU ..... Rectified Linear Units

1 Introduction

In this project different data science techniques are applied to develop data assessment tools and predictive models that are used to predict the fatigue strength of steel. Although some theoretical methods are used for fatigue analysis, their implementations are time consuming. To predict the fatigue strength, the fatigue dataset for steel from Agrawal et al.[1] is used. This dataset is originally from National Institute for Material Science (NIMS) public domain database. The dataset consist of chemical compositions and processing parameters of three different grades of steels namely, carbon and low alloy steels, carburizing steels and spring steels. There are in total 437 samples/rows, 25 features/columns, and 1 target feature (fatigue strength). Among this 437 samples, 371 are carbon and low alloy steels, 48 carburizing steels, and 18 spring steels. This data pertains to various heats of each grade of steel and different processing conditions. In this 25 features, there are 9 chemical compositions (in wt.%), 16 processing parameters and one mechanical property i.e., Rotating Bending Fatigue Strength (10^7 Cycles). The details of the 25 features are given in Table 1. The main purpose of this project is to implement four machine learning(ML) algorithms from scratch, namely, K-mean Clustering, Multi Linear Regression(MLR), Neural Network(NN) and K Nearest Neighbors(KNN) to predict the fatigue strength of steel and other mechanical properties.

2 Theory

2.1 Scientific background

To create a correlation between different properties of alloys along with their chemical compositions and manufacturing process parameters one requires a physics based or data driven approach. Here data driven approaches are of significant interest to material scientists as the latest physics based models have severe limitations when it comes to computing extreme value properties like cyclic fatigue. This approach refers to the combination of data science and analytics which is expected to offer an distinct set of tools for an integrated approach and establish a desired relations like process-structure-property (PSP) linkages which is based on the observation made from controlled experiments and advanced physics based models. In the recent years experts in materials science have identified the set of definite tools by integrating three fundamental components i.e., experiments-models-data analytics and establishing the desired relations. Data science and analytics is anticipated to have an positive impact on materials development by maximizing the accuracy and the reliability of the key insights extracted from a large collection of dataset[1]. In fact, materials data and ML provides the foundation for this data-driven materials discovery paradigm, which integrates materials domain knowledge and artificial intelligence technology to form the new research field of materials informatics(MI). In this new field, the Materials Genome Initiative aims to halve the cost and time for discovery to development to deployment of advanced materials[2]. Progress in this field is complemented by the availability of large amounts of experimental and simulation data alongside with enhanced data analytics tools. This integrated approach uses materials data to investigate structure-property relationships and to develop predictive models which act as a guide for synthesis of new materials[2].

2.2 Scientific work

In this project a systematic framework is created by establishing reliable linkages between process variables in a class of steels, their chemical compositions, and their fatigue strengths. As seen in Figure 1, the approach in this work is structured into four steps: (i) Data preprocessing,

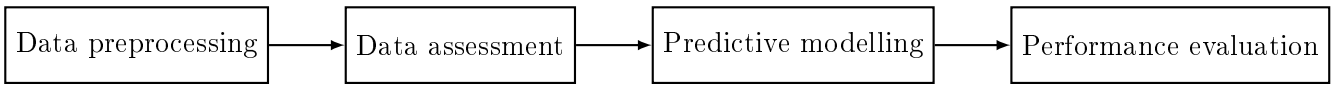


Figure 1: Work flow to predict the fatigue strength of steel

to transform raw input data into a readable and understandable format; (ii) Data assessment, to check there exists any pre-defined pattern and create feature ranking based on correlation with fatigue strength(target or dependent feature); (iii) Predictive modelling using three different ML models; (iv) Performance evaluation of the models based on prediction accuracy. The accurate prediction of the fatigue strength of steel is of a major interest due to the cost and time required for fatigue testing and the consequence of fatigue failures. Fatigue strength is the basic data required for the design and failure analysis of structural components as fatigue accounts for over 90% of all mechanical failures[1]. Thus, predicting fatigue life is of great importance for material scientists and engineers. The scope of this work includes four machine learning models and data analytics tools to predicit fatigue strength of steel using composition and processing parameters. This leads to the identification of structure-property relation which is the fundamental for new materials discovery and development of anti-fatigue high strength steels.

Table 1: 26 features of the 437 NIMS fatigue data[1]

Features	Description	Min	Max	Mean	StdDev
C	% Carbon	0.17	0.63	0.388	0.096
Si	% Silicon	0.16	2.05	0.3	0.246
Mn	% Manganese	0.37	1.6	0.823	0.279
P	% Phosphorus	0.002	0.031	0.016	0.005
S	% Sulphur	0.003	0.03	0.015	0.006
Ni	% Nickel	0.01	2.78	0.517	0.853
Cr	% Chromium	0.01	1.17	0.57	0.412
Cu	% Copper	0.01	0.26	0.068	0.049
Mo	% Molybdenum	0	0.24	0.07	0.088
NT	Normalizing Temperature	825	930	872.3	26.212
THT	Through Hardening Temperature	30	865	737.643	280.037
THt	Through Hardening Time	0	30	25.95	10.264
THQCr	Cooling Rate for Through Hardening	0	24	10.654	7.841
CT	Carburization Temperature	30	930	128.856	281.744
Ct	Carburization Time	0	540	40.502	126.925
DT	Diffusion Temperature	30	903.333	123.7	267.129
Dt	Diffusion time	0	70.2	4.844	15.7
QmT	Quenching Media Temperature (for Carburization)	30	140	35.492	19.419
TT	Tempering Temperature	30	680	536.842	164.102
Tt	Tempering Time	0	120	65.08	21.478
TCr	Cooling Rate for Tempering	0	24	20.815	8.072
RedRatio	Reduction Ratio (Ingot to Bar)	240	5530	923.629	576.617
dA	Area Proportion of Inclusions Deformed by Plastic Work	0	0.13	0.047	0.031
dB	Area Proportion of Inclusions Occurring in Discontinuous Array	0	0.05	0.003	0.008
dC	Area Proportion of Isolated Inclusions	0	0.058	0.008	0.01
Fatigue	**Rotating Bending Fatigue Strength (10^7 Cycles)**	**225**	**1190**	**552.904**	**186.631**

3 Numerical methods and Implementation details

3.1 Project design

In this project a structural framework is devised to predict the fatigue strength of steel. As seen in Figure 2, the entire project is programmed in this structure and the details of each process will be discussed in the coming sections. This design framework is generalized to predict not only fatigue strength but also other mechanical properties or dependent features when provided with a dataset that satisfies the required conditions.

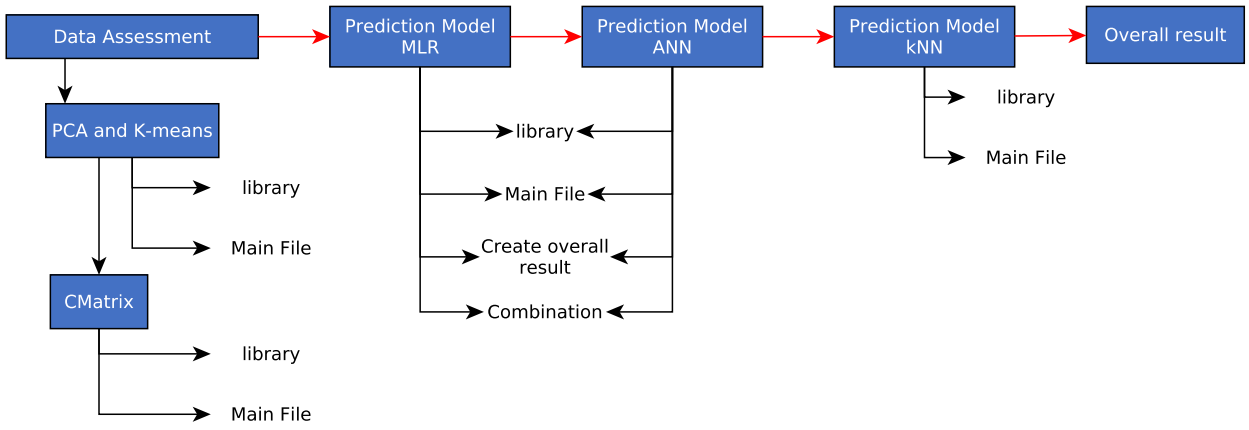


Figure 2: Structural framework of the implemented models.

3.2 Data assessment

To assess the input dataset three different data assessment tools are applied, namely, Principal component analysis(PCA), K-mean clustering and Correlation matrix. In PCA the d dimensional input dataset is reduced into a two-dimensional data (i.e., into two data with high variance) and assessed if the details of original dataset is retained in the extracted one. In K-means, the clusters are made form the reduced data and analysed if the clusters from the original dataset still exist. In correlation matrix, correlations between all features are identified and also used to verify the findings of PCA. The details of the features implemented in the data assessment step can be found in Table 2. The directories and results generated during data assessment step can be referred from Table 3.

3.3 Principal Component Analysis

The visualization and analysis of a d dimensional data with n observations/samples becomes more challenging as our ability to visualize data is limited to 2 or 3 dimensions. We may explore these datasets using two-dimensional scatter plots, each of which contains n observations and two of the d features. If in case the dataset has $d = 15$ features then there are 105 plots to draw and it is very likely that none of them will be informative since they each contain just a small fraction of the total information present in the dataset[3]. Hence, we require a low-dimensional($k \ll d$) representation of the data that captures as much of the information as possible. One technique that provides such a dimensionality reduction is Principal Component Analysis (PCA), which projects a high-dimensional feature space onto a new feature space. The original explanatory features are replaced with new features called principal components. These features are derived from the original features, and are by design uncorrelated

Table 2: (a)Functionalities implemented in PCA and K-means Clustering; (b)Functionalities implemented in Correlation matrix

(a)Functionality Implemented	Remarks	(b)Functionality Implemented	Remarks
Import dataset	File format: .csv	Import dataset	File format: .csv
Feature scaling	Standardizing features	Feature scaling	Standardizing features
Feature extraction	PCA; Write select PC results; Creates Scree plot and Biplot; Writes extracted features in format: .csv	Correlation matrix	Pearson correlation; Creates a heatmap with all correlations
Elbow method	Within-Cluster Sum of Square (WCSS); Creates elbow plot	Feature selection	Creates correlation based feature ranking and writes selected features into a file in the format: .csv
K-means	Creates cluster and parallel coordinate plot; Writes K-means results	-	-

to one another, thus eliminating the redundancy. The main idea behind PCA is that not all of the d dimensions of the original data set are equally informative, where the concept of being informative is measured by the variability along each particular feature space dimension, also denoted as variance[3]. More precisely, PCA finds the directions of maximum variance in high-dimensional data and projects it onto a smaller dimensional subspace while retaining most of the information. Each of the dimensions found by PCA is effectively a linear combination of the d features[3]. There also exist other feature extraction algorithms like Singular Value Decomposition(SVD) which is similar to PCA and Linear Discriminant Analysis(LDA) which is used for supervised algorithms where PCA hampers the data. The PCA and K-means are implemented together and programmed into two files as seen in Figure 2. Here, one file work as a library implemented with required functions and the other is the driver which performs data preprocessing, feature extraction and clustering. More details on the implemented features will be discussed in the upcoming sections.

3.3.1 PCA Data preprocessing

Preprocessing of the dataset is the first step of implementation in which two functions are defined, one for importing the dataset(*import_ dataset*) and other for standardizing(*feature_ scaling*) the dataset using feature scaling technique. The purpose of this preprocessing step is to transform raw input data into a readable and understandable format. Before importing the dataset the user should check whether the dataset complies to the specified set of conditions which is applicable to the entire project. The *import_ dataset* function reads the input dataset and separate independent features(X) from the dependent feature(y) and returns X and y. The next step of data preprocessing is to standardize these X and y features into a uniform scale. The *feature_ scaling* function is used to perform standardization using Equation 1. Here, z is the standardized version of the feature(which will be X or y), x is the input feature, μ is the mean of the input features and σ is the standard deviation of the input features. To check the correctness of the implementation the mean value of the scaled data must be close enough to zero and the standard deviation must be close enough to one. We scale both the features and return them as output but we ignore scaled dependent feature as PCA and Kmeans are unsupervised models. This data preprocessing step will be almost similar for other upcoming algorithms with some added functions.

$$z = \frac{x - \mu}{\sigma} \tag{1}$$

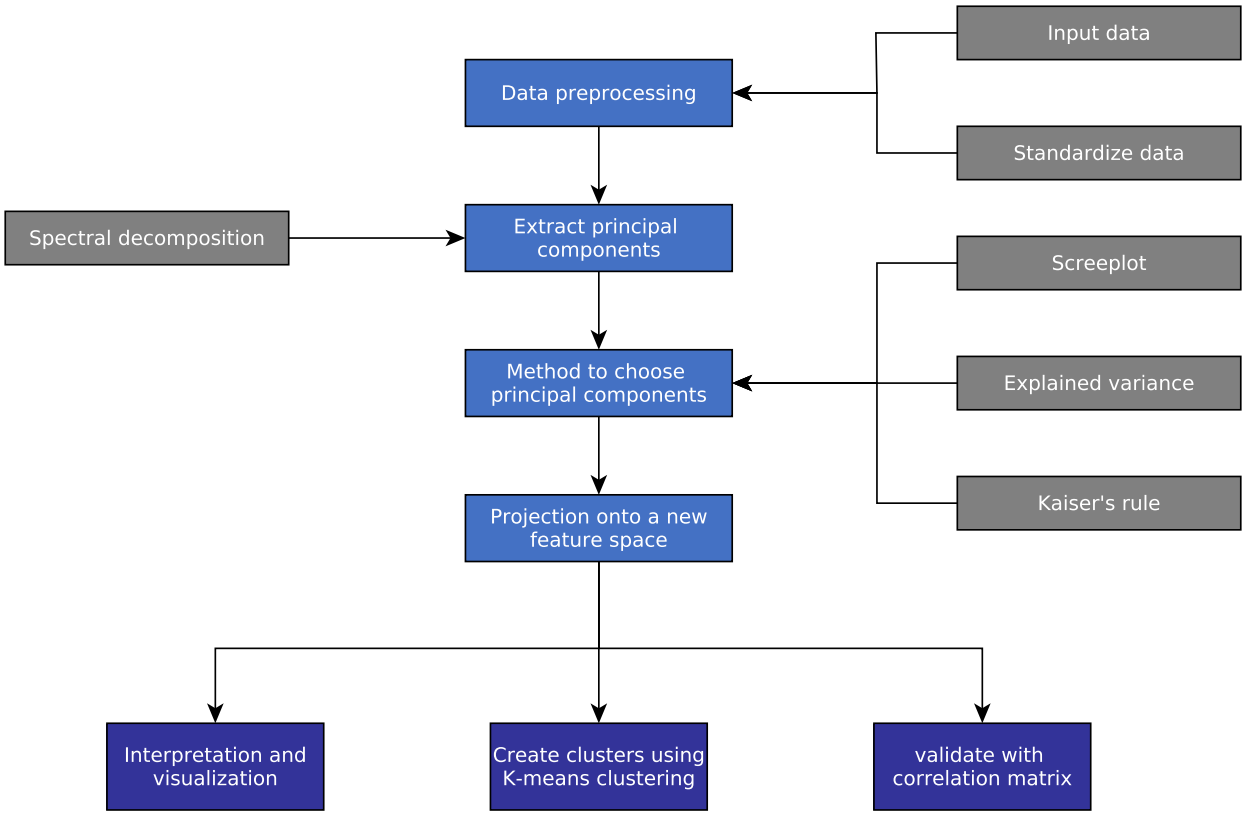


Figure 3: Work flow of principal component analysis[3].

Set of conditions that input dataset must comply:

- In the *import_dataset* function the input data are read in as pandas.read_csv, thus, all input datasets that are imported should be in the format .csv.
- The data entries present in the dataset should be real numbers and the empty ones should be filled based on domain knowledge or zeros.
- In a dataset there must exist atleast one independent feature and the dependent feature must be the last column of the dataset.

3.3.2 Extract principal components

The core principle in PCA is that the principal components are obtained from the eigenvectors of the covariance matrix. The eigenvectors (principal components) determine the directions of the new feature space, and the eigenvalues gives the variances of their respective principal components. The popular approaches to compute the principal components is referred to as the spectral decomposition or eigendecomposition[3]. To understand this clearly extraction will be explained together with the implemented code section as specified in Listing 1. The data covariance matrix is computed by numpy.cov, in which the scaled independent dataset is given as input. From the covariance matrix eigen values and eigen vectors are computed by scipy.linalg.eigh. The eigen vector with the highest eigen value is the first principal component of the dataset. Consequently, once eigenvectors are found from the covariance matrix, the next step is to order them by eigenvalue, highest to lowest[3]. To sort the eigen values and vectors from maximum variance to minimum numpy.argsort is used. From the sorted eigen vector only the first k eigen vectors are selected based on the magnitude(maximum) of eigen values. By

neglecting some principal components we lose some information, but if the eigenvalues of the neglected ones are small, we do not lose too much. The goal is to find the smallest number of principal components required to get a good representation of the original data[3].

```
import numpy as np
import scipy.linalg as spl
# Calculating the covariance of the scaled data;
# Size ->[independent features x independent features]
# rowvar = False -> each column represents a variable,
# while the rows contain observations
mat_Covariance = np.cov(scaled_X, rowvar = False)
# Calculating the eigen values and the eigen vectors of the mat_Covariance
eigen_val,eigen_vec = spl.eigh(mat_Covariance)
# eigen vector; size -> [independent features x independent features]
# eigen value; size -> [independent features]
# Sorting eigen values and eigen vectors
# Sorting in descending to get max variance -> min variance
# and returning their index
sort_index = np.argsort(eigen_val)[::-1]
# Eigen value sorted from max variance to min
self.sort_eigen_val = eigen_val[sort_index]
# Eigen vector sorted from max eigen vector and its magnitude
# will be the max variance
self.sort_eigen_vec = eigen_vec[:,sort_index]
# Extracting the features w.r.t chosen principal components
# Size -> [independent features x selected no of PC]
self.extracted_eigen_vec = self.sort_eigen_vec[:,0:selected_PC]
# Projecting n-dim dataset into 2-dim feature space
# Extracted features; Size -> [No of samples x selected no of PC]
extracted_X = np.dot(scaled_X,self.extracted_eigen_vec)
return extracted_X,self.sort_eigen_val
```

Listing 1: Python implementation of Principal Component Analysis

3.3.3 Choose principal components

As we know the data used in this project is the fatigue strength dataset with 25 features and 437 samples. To reduced the dimension of the dataset from 25 features to k features and to select the number of principal components, three different methods are implemented. Usually first two principal components are selected as they have the highest variance and can be interpreted by creating scatter plots. But, there is no standard way to decide how many principal components are enough for a given dataset[3]. However, there are three simple approaches which may be of guidance for deciding the number of relevant principal components. These are, (i) the visual examination of a scree plot; (ii) the variance explained criteria; (iii) the Kaiser rule [3].

The visual examination of a scree plot: The Scree plot as depicted in Figure 5(a) has 25 principal components as per 25 features and plotted against the proportion of variance explained by each principal components. The proportion of variance is computed by simply dividing the variance(eigen values) explained by each principal component by the total variance explained by all principal components[3]. A widely applied approach is to decide on the number of principal components by examining a scree plot. By eyeballing the scree plot, and looking for a point at which the proportion of variance explained by each subsequent principal component drops off. This is often referred to as an elbow in the scree plot[3]. For the dataset used the drop off is observed in the scree plot at third principal component. Thus, based on the scree plot we decide to pick first three principal components to represent our extracted dataset, thereby explaining 60% of the variance as seen in method1 results(marked with red) from Figure 4.

```
Select any one method from below to choose number of principal components to extract:

Method1: The varience explained criteria

[0.41105724 0.5415641  0.60535186 0.66321353 0.7192615  0.76744839
 0.80932038 0.8428735  0.87088971 0.89742672 0.91889451 0.93799449
 0.95237407 0.96508972 0.97654189 0.9869686  0.99209789 0.99679678
 0.99869286 0.99948625 0.99998864 0.99999525 1.         1.
 1.        ]

Recommended number of principal components to select = 7

Note: The number of principal components are selected w.r.t a threshold of 80%(0.80)

#-----------------------------------------------------------------------

Method2: Kaiser-Guttman criterion(frequently used)

[10.30000072  3.27015472  1.59835164  1.44985937  1.40441304  1.20743543
  1.04920067]

Recommended number of principal components to select = 7

Note: Principal components with varience > 1 are selected

#-----------------------------------------------------------------------

Method3: Scree plot

Note: Select from plot_Scree.png present in the directory Plots_KMeans
```

Figure 4: Results of three different methods implemented to choose the number of principal components.

The varience explained criteria: Another simple approach to decide on the number of principal components is to set a threshold by 80%, and stop when the first k components account for a percentage of total variation greater than this threshold[3]. For the project dataset the first seven components account for 80.9% of the variation and the same can be seen in Method1 results(marked with green) of Figure 4. Thus, based on the variance explained criteria we pick the first seven principal components to represent our extracted dataset. Note that the threshold is set somehow arbitrary; 70 to 90% are the usual sort of values, but it depends on the context of the data set and can be higher or lower[3].

Kaiser's rule (Kaiser-Guttman criterion): The Kaiser's rule is a widely used method to evaluate the maximum number of linear combinations to extract from the data set. According to the rule, only those principal components whose variances exceed one are retained. The idea behind the Kaiser-Guttman criterion is that any principal component with variance less than one contains less information of the original variables and thus, not worth retaining[3]. For the fatigue dataset, the Method2 results as seen in Figure 4, shows that the first seven components has variance greater than one. Thus, based on the Kaiser's rule we pick the first seven principal components to represent our extracted dataset. From the above discussed methods it clear that for the project dataset the number of principal components to select is seven and same is written into the extracted dataset using pandas.DataFrame.to_csv for analysis. In the implementation point of view the number of principal components are selected based on user inputs as seen in second row of Table 4. The user can select the input based on the results presented in Figure 4 or from the recommendations provided. By default first two principal components(PC) are selected as it will have the maximum variance and atleast two PC are required to perform analysis. So the minimum input expected from the user is two and if it is below two, the program exits with an error message: "Error: *selected_pc* input must be >= 2". The user is also recommended to use Kaiser-Guttman criterion as it is considered as an reasonable method.

3.3.4 Derive new features

After selecting the number of principal components the dimension of the original data is reduce by projecting the data onto a new feature space with less dimensions. This reduces our original dimension of 25 feature into selected no of principal components(two). In PCA the extracted eigen vectors are called as the loading vector and the elements in the vector are called as loading. For visualization and analysis purpose we select the maximum variance i.e., the first two principal components with a variance explained criteria of 54%. The mapping of the n observations, in this case, 437 observations with 25 features into a new space results in same observations but with just two features. This is done by taking a dot product between the observation and the extracted eigen vector and the same can be observed in line 26 of Listing 1. These new values are denoted as scores and are the projection of the original data onto a new feature space. The detailed analysis of the extracted scores will be discussed in the result analysis section(Sect.4.2.1).

3.4 K-means Clustering

After reducing the dataset from d dimension to k dimension, K-means clustering algorithm as in Figure 6 is applied to check whether the groups or clusters existed in the original dataset is still present in the extracted dataset or score dataset. In addition to that, K-means can also be applied to any two-dimensional dataset to create cluster plot by finding suitable number of clusters(centroids). In this implementation suitable number of clusters are identified from the elbow plot created using elbow method. The K-means clustering is an iterative algorithm that is used to divide the entire dataset into subgroups based on the selected number of centroids. This is an unsupervised machine learning algorithm as there does not exist any target feature and are grouped completely based on the similarities present in the dataset. The group of data points(samples) present in a cluster are similar to each other, whereas, the clusters themselves are as different as possible. The objective of this algorithm is to make sure that the sum of the distance between the data points and their respective clusters centroid is at the minimum. This is computed using squared error function as given in Equation 2[4]. The results of the clusters are then analysed using parallel coordinates plot to check whether the created clusters resemble the characteristics of the original dataset. More details on the implementation front will be discussed in the upcoming sections.

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \left\| x_i^{(j)} - c_j \right\|^2 \tag{2}$$

Here, J is the squared error function, k is the number of clusters, n is the number of samples, x_i is the i^{th} sample and c_j is the centroid of the cluster j.

3.4.1 Working of K-means algorithm

In K-mean clustering algorithm clusters are created based on the selected number of clusters. Each cluster will have its own centroid and it will be the mean center of the data points present in a cluster. Initially these centroids are selected randomly from the dataset using pandas.DataFrame.sample. For this implementation, the reduced dataset from PCA will be the input dataset and the number of cluster shall be three. For three clusters, three random data points(at $random\_state = 0$) from the dataset are selected as centroids. The data points which are closer to each centroids group together and form clusters. To find which cluster does the i^{th} data point belongs to, the distance between the i^{th} data point and the j^{th} centroid of all k clusters are computed. In this case the distance between each data point and centroid of

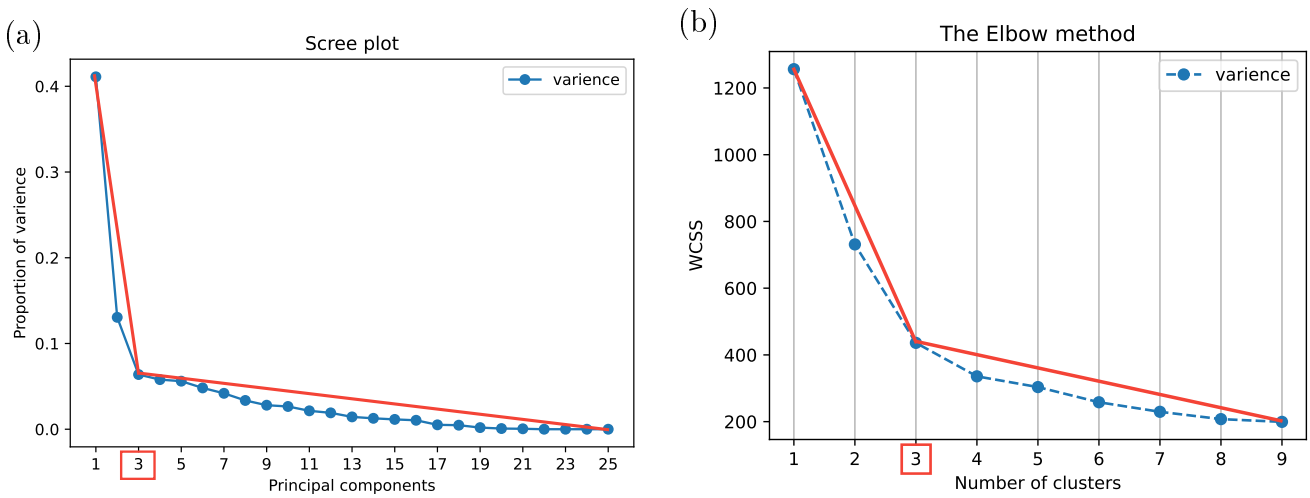


Figure 5: (a)Scree plot determining the number of principal components as three; (b)Elbow method plot determining the number of clusters as three.

three clusters are calculated. The data point will be then assigned to the cluster whose centroid it is closer to. When the distance between the data point and the centroid is calculated, an if statement is used to find which centroid the data point is closer to. A minimum distance is preinitialised as a large value, in this case an infinity. In the first iteration the if statement checks whether the minimum distance is greater than the distance between the first centroid and the data point. The condition will be satisfied as the minimum distance is infinity, and now, the calculated distance will be assigned as the minimum distance. The distance between the data point and the next(second) centroid is computed. Once again the if statement checks whether the minimum distance i.e the distance between the data point and the first centroid is greater than the second, if yes, the second will now be the minimum distance. This is repeated for all j centroids of the k clusters and the last standing centroid, which is the minimum distance, will be the centroid whose cluster the data point belongs. This is repeated for 437 samples and three different clusters are made. The details of which cluster each data point belongs is noted. The aim of the algorithm is to have a minimum distance between the sum of the data points present in a cluster and its centroid. To do so, we compute the mean of all data points present in each cluster, likewise, for three clusters and check the difference between the computed mean values and the centroid values. The mean is computed as seen in Listing 2(a), where the extracted dataset from PCA is converted into a dataframe and the data points present in the dataframe are grouped using pandas.DataFrame.groupby with respect to the noted cluster details. As we have considered three clusters, three groups will be formed and the mean of each group is computed. If the computed difference between the mean values and the centroid values is zero, then the distance between sum of the data points and the centroid in a cluster is at its minimum, the condition is satisfied. If not, the computed mean will be considered as the new centroid and the process will be repeated until this condition is satisfied.

3.4.2 Choose number of clusters

In the previous section we considered the number of clusters to be three, but to find the optimal number of clusters for a given dataset the Elbow method is used. In this method a predefined set of clusters is assigned for which Within-Cluster Sum of Square(WCSS) is computed as seen in Listing 2(b). WCSS is the sum of squared distance between each data point and the centroid in a cluster. The predefined set of cluster is the number of cluster values that is ranging from one to end limit. This end limit is an user input and by default it is assigned as nine and the

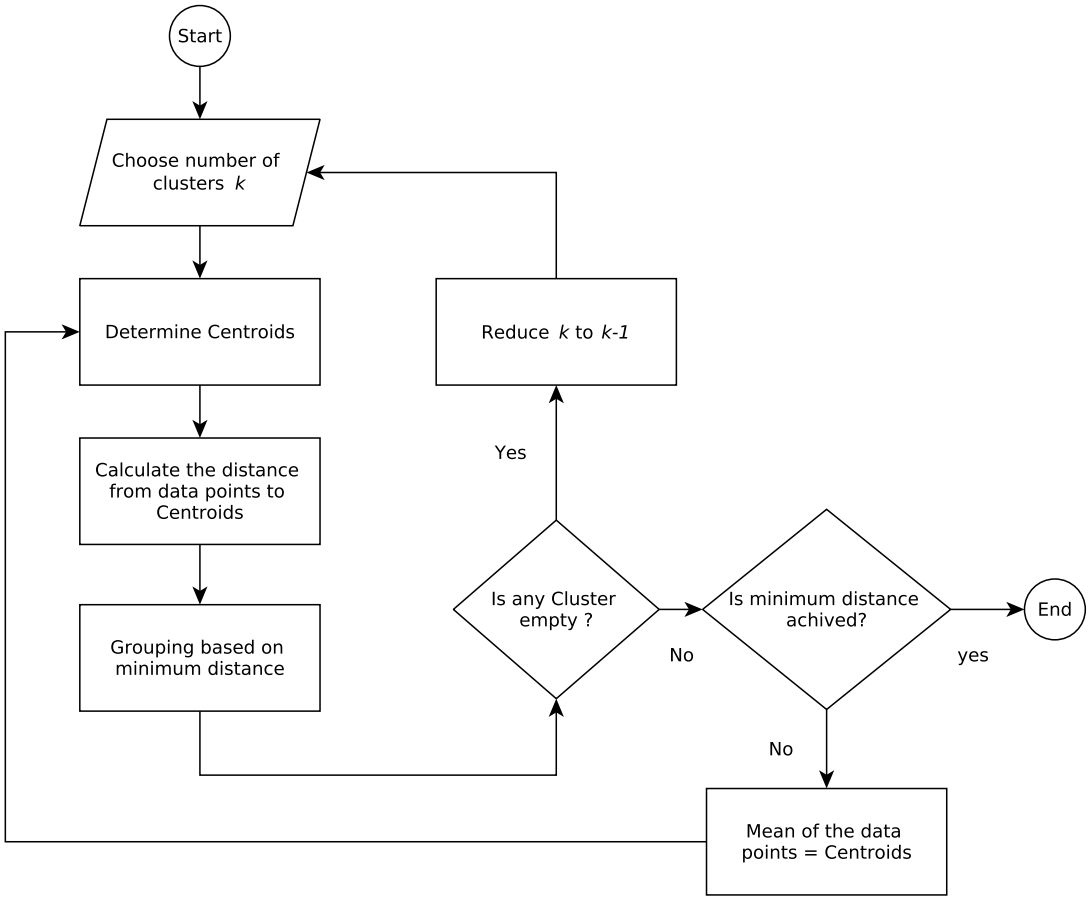


Figure 6: Flowchart explaining the working of the K-means Clustering algorithm.

same can be seen in fourth row of Table 4. The user input of the end limit also referred to as $kfind$ in click should be greater than zero, if not the program exits with an error: "Error: $kfind$ input must be > 0". To compute WCSS for each number of clusters one needs the centroid and cluster details of all the number of clusters present in the predefined set. That is for each j cluster from one to ten, K-means clustering algorithm is applied to make clusters. From the created clusters one can get the details of data points present in the cluster and the centroid of the cluster. This is then used to compute WCSS value for each number of clusters. The elbow plot is constructed using the set of predefined clusters and its respective WCSS values, and the same is depicted in Figure 5(b). From the plot we can observe that, as number of clusters increase the WCSS value decreases, because, as the number of clusters increases more clusters are made and thus, the distance between the data points and the centroid in each cluster decreases, as a result WCSS decreases. To select the number of clusters for a given dataset we should look at the point where the elbow shape is observed, and after that point the plot should start to be parallel to the x-axis. This observed point is considered to be the optimal number of clusters for a given dataset. In case of the reduced dataset, the point at which the elbow shape is observed is three. Thus the number of clusters selected will be three and for the same the cluster plot is created and the results of which will be analysed.

Important detail to note during WCSS computation: It is not always the case that clusters are possible for the input number of clusters(k), as the algorithm is looking to compute the minimum distance between sum of the data points and the centroid in a cluster. For example, if the number of clusters are five, the K-means algorithm is applied to create five clusters with each clusters having its own centroid. To check whether the minimum distance

is attained, the mean values of the five clusters are computed and checked with the existing centroid values. If the condition does not satisfy the mean value will be assigned as the centroid and the entire process is repeated. During this re-computation step there is a possibility that non of the data points are closer to the fifth centroid and thus the centroid has no cluster. In this case the program exits with an error message: "Retry: Clusters are only possible till $k = k - 1$. Update $kfind$ to $k - 1$", where $k - 1$ is the possible end limit($kfind$) value for the defined cluster set.

```
# extracted_Xdf - Extracted dataset
Centroids_update =  extracted_Xdf.groupby(Clusters).mean().values #-----(a)
#-------------------------------------------------------------------------
# Function for WCSS Calculation
def calculate_WCSS(extracted_X,Clusters,Centroids):                #-----(b)
  sum_distance = 0  # Initially assigning sum to be zero
  for i,val in enumerate(extracted_X):
    sum_distance += np.sqrt((Centroids[int(Clusters[i]),0] - val[0])**2 +
     (Centroids[int(Clusters[i]),1] - val[1])**2)
  return sum_distance
```

Listing 2: Python implementation for computing the mean of grouped data points(a) and for computing WCSS value(b).

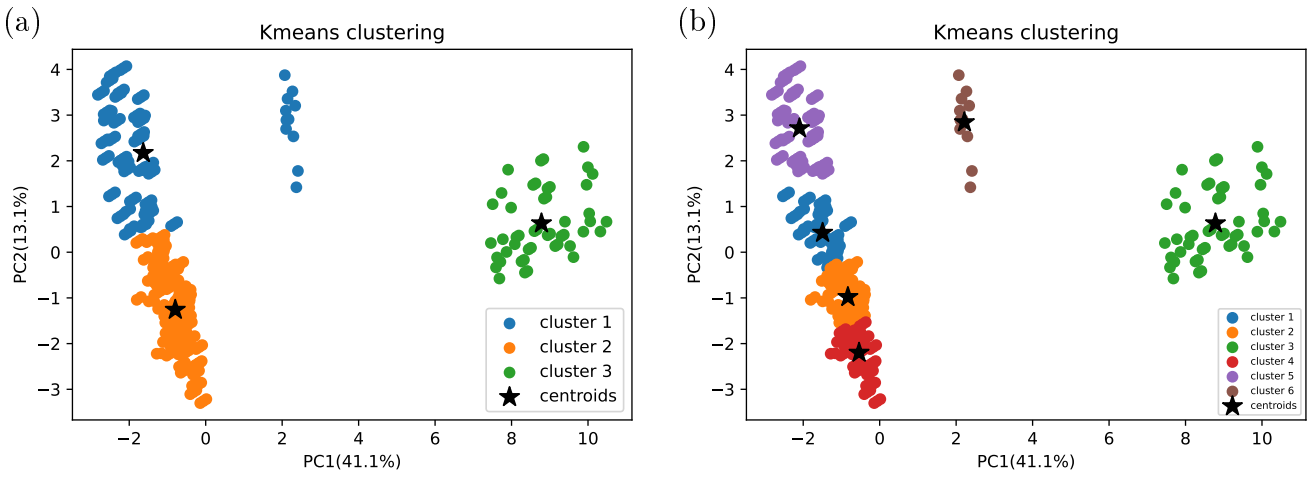


Figure 7: (a)K-means clustering of the reduced dataset at number of clusters(k) three; (b)K-means clustering of the reduced dataset at number of clusters(k) six.

3.4.3 Make clusters

The number of clusters(k) determined from the elbow plot for the reduced dataset is three. As we know the original dataset consists of chemical composition and processing parameters of three different grades of steels namely, carbon and low alloy steels, carburizing steels and spring steels. This means the original dataset already have three clusters and the same is confirmed on the reduced dataset using elbow method. In the implementation front, the clusters are computed at number of clusters three in the same way as specified in the working of K-means. Additionally, the details of distance computation, i.e., the distance between the i^{th} data point and the j^{th} centroid of all k clusters are stored into a dataframe together with the details of final centroids, reduced dataset and clusters, the same shall be observed in Figure 8. The number of clusters used to create the final clusters are based on user input of k in click and the same can be observed in third row of Table 4. The recommended input will be the clusters observed from the elbow plot and the input must be greater than zero, if not the program exists with

an error message: "Error: k input must be > 0". In order to visualize the created clusters, a clustering plot is made as seen in Figure 7(a), in which each clusters created are plotted in different colour and the centroids of the same is represent as a star in color black. Just for visualization purpose another cluster plot is created as seen in Figure 7(b) at number of clusters six. Interpretation and validation of the plotted results are discussed in more details on the result analysis section(Sect.4.2.2).

```
Resultant centroid coordinates of the cluster:

X_coordinate: [-1.63279442 -0.79521455  8.78279505],
y_coordinate: [ 2.17135774 -1.25991919  0.63161364]

Resultant dataset:

+------------+------------+------------------+------------------+------------------+-----------+
|        PC1 |        PC2 |   Dist_Centroid1 |   Dist_Centroid2 |   Dist_Centroid3 |   Cluster |
+============+============+==================+==================+==================+===========+
|  2.10257   |  2.69607   |         3.77204  |         4.90377  |         6.99195  |         1 |
+------------+------------+------------------+------------------+------------------+-----------+
|  2.3719    |  1.421     |         4.07438  |         4.14945  |         6.45932  |         1 |
+------------+------------+------------------+------------------+------------------+-----------+
|  2.29448   |  2.53412   |         3.94399  |         4.89294  |         6.76149  |         1 |
+------------+------------+------------------+------------------+------------------+-----------+
|  2.16865   |  2.90801   |         3.87217  |         5.11431  |         6.99492  |         1 |
+------------+------------+------------------+------------------+------------------+-----------+
|  2.34179   |  3.20571   |         4.10697  |         5.45735  |         6.93631  |         1 |
+------------+------------+------------------+------------------+------------------+-----------+
```

Figure 8: The first five entries of the K-means clustering results computed using extracted dataset at number of clusters(k) three.

Table 3: Files and directories created during PCA, K-means and CMatrix implementation

Directory	Files and plots	Remarks
Plots_KMeans	plot_Scree.png plot_ElbowMethod.png	Plots to select no of PC and no of clusters as seen in Figure 5
-	plot_KmeanClustering.png	K-means clustering plot as seen in Figure 7
-	plot_PCA.png plot_parallelCoordinates.png	Biplot to analyse PCA and parallel plot to analyse K-means as seen in Figure 26, Figure 29
Results_KMeans	SelectPrincipalComponents.txt LoadingResults.txt	Results of select no of PC as seen in Figure 4; Loading vector and value as seen in Figure 27
-	fatigue_Extracteddataset.csv resultKMeans.txt summaryKMeans.txt	Extracted dataset w.r.t select no of PC; Results of Clustering and K-means result summary as seen in Figure 8, Figure 28
Plots_CMatrix	plot_AllCorrelation.png plot_Correlationwith_target.png	Correlation plot of all features as seen in Figure 9; Correlation plot of feature ranking w.r.t target feature as seen in Figure 10
Results_CMatrix	fatigue_Selecteddataset.csv	Feature selected dataset based on user input

Generalization of PCA, K-mean algorithm and other details:

- The PCA and K-mean algorithms are design to work for any given dataset that complies the mentioned set of conditions.

- As a part of generalization **click** library is used as seen in Table 4 to adjust certain user defined inputs. These inputs can be given in the command line while executing the program, if not given, default values are set for every input and same can be seen in Table 4. To get more information on input options, try *filename.py - -help*.
- Each time when a plot is created or a file is written the program automatically creates a directory if one does not exist and store the results and the same can be found in the Table 3.
- The program must be executed from the driver file *PPP_KMeans_main.py* as all created features are already imported from the library *PPP_KMeans.py* and the input datasets must be present in the same directory as the programming files.
- This confirms the completion of the task K-means Clustering as specified in the proposal. PCA implementation is additional to what was proposed.

Table 4: Python implementation of click in PCA and K-means with default values and the required input data types

click.option	nargs	dtype	default	help
- -data	1	str	fatigue_dataset.csv	Enter input dataset .csv: last column must be the target feature
- -selected_pc	1	int	2	Select no of PC for extraction from Figure 4
- -k	1	int	3	Select no of clusters from Figure 5(b)
- -kfind	1	int	9	Enter end limit for no of clusters in Figure 5(b)
- -target_column	1	str	Fatigue	Enter Target column header from dataset
- -plt_size	4	float	[7.5, 5, 7.5, 5]	Enter plot size for Figure 5(a) and Figure 26

3.5 Correlation Matrix

The Correlation matrix (CMatrix) is a feature selection technique that is used to determine the correlations between all the features present in a dataset. The created correlations are then used to select highly correlated features and to verify the results obtained from the biplot of PCA as seen in Figure 26. The Pearson correlation coefficient also known as the bivariant correlation is a measure of linear correlation between two sets of data as seen in Equation 3[5]. It is the ratio between the covariance of two variables and the product of their standard deviation. This correlation is only suitable for quantitative features and the results are always between the values -1 and 1. In case of more than two features the correlation is computed between all different feature pairs and results a square table known as the correlation matrix. In the implementation the correlation matrix is computed using pandas.DataFrame.corr and plotted as a heatmap using **seaborn** library. The fatigue dataset used in the project consist of 26 features, the correlation matrix created using these features result in a [26 x 26] matrix and the same is depicted in the Figure 9. From Figure 9, one can observe that all the diagonal elements are one as the feature correlates with itself. Additionally, the correlations beneath the diagonal are redundant as it is a symmetrical matrix. There also exists other correlations techniques,

namely, Spearman correlation and Kendall's tau, but both are suitable for ordinary variables [5]. The CMatrix is programmed into two files with one acting as a library implemented with the required functions and the other is the driver which creates correlation plot and writes selected features into a csv. More details on the implemented features will be discussed in the upcoming sections.

$$r_{XY} = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}} \tag{3}$$

Here, r_{XY} is the correlation coefficient, X_i, Y_i is the value of X and Y variable in a sample and $\overline{X}$, $\overline{Y}$ are the mean values of X and Y variable.

Degree of correlation:[6]

- **Perfect:** If the value is near ± 1, then it said to be a perfect correlation: as one variable increases, the other variable tends to also increase (if positive) or decrease (if negative).
- **High degree:** If the coefficient value lies between ± 0.50 and ± 1, then it is said to be a strong correlation.
- **Moderate degree:** If the value lies between ± 0.30 and ± 0.49, then it is said to be a medium correlation.
- **Low degree:** When the value lies below ± 0.29, then it is said to be a small correlation.
- **No correlation:** When the value is zero.

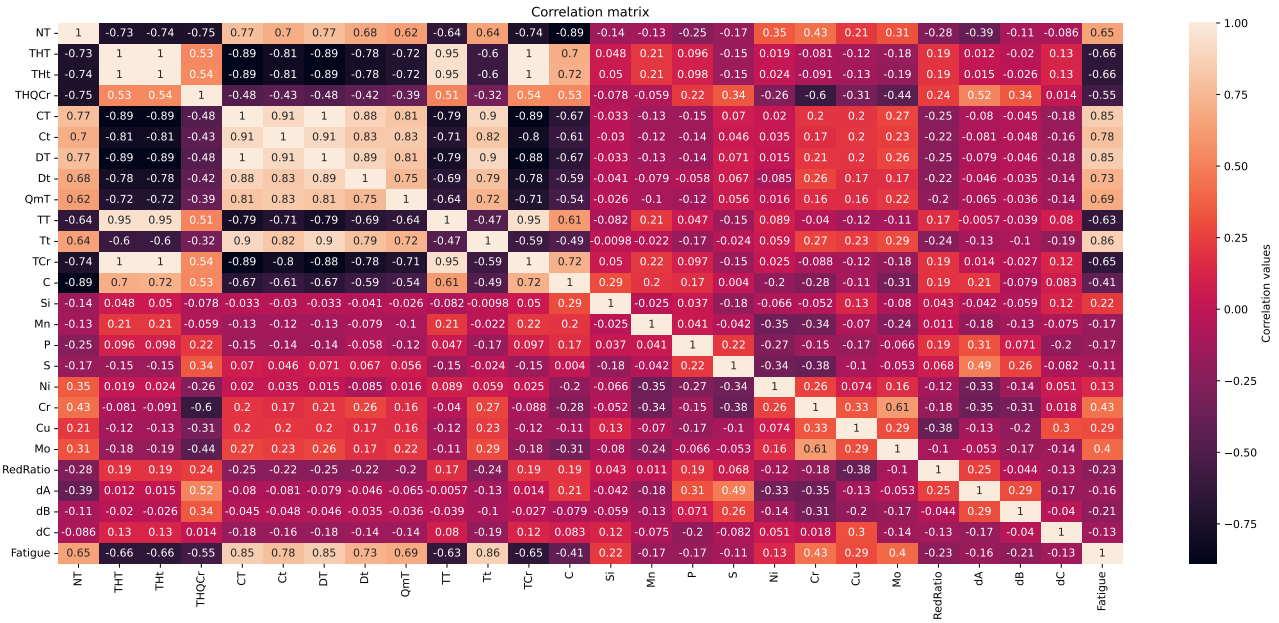


Figure 9: The Correlation matrix of all 26 features with pair wise correlation with each other. The abbreviation for the features in x and y axis shall be referred from Table 1.

3.5.1 CMatrix Data preprocessing

The data preprocessing step in CMatrix is similar to what was discussed in PCA(Sect.3.3.1). In the implementation, the *Data_preprocessing* class consist of two functions, one to read in the input dataset as a dataframe and other to perform feature scaling. The *import_dataset*

function reads the input dataset and returns a dataframe without any changes. The *feature_scaling* function standardize the dataframe using Equation 1. This scaled dataset with the uniform format will be used as an input to create correlation matrix.

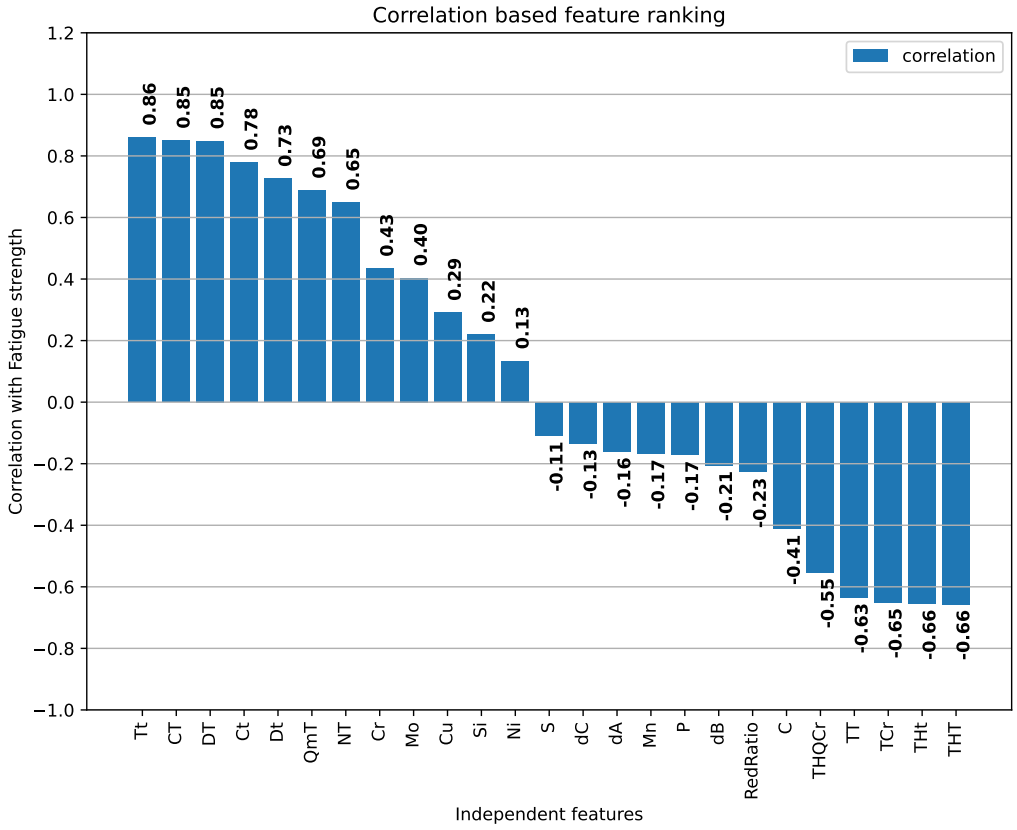


Figure 10: Correlation based feature ranking of all the features with fatigue strength from positive to negative correlation. The abbreviation for the features in x axis shall be referred from Table 1.

3.5.2 Feature selection

Feature selection is a preprocessing step that is used to effectively reduce the dimensionality of the dataset by removing irrelevant and redundant features. In this project feature selection is done by feature ranking the correlations between the features and the target variable. In the fatigue dataset there are 25 features and 1 target feature. The correlation between the 25 features and the fatigue strength is filtered and then ranked from high positive correlation to high negative correlation as depicted in Figure 10. Based on the ranking, features with positive correlations are selected, and a selected dataset is written using pd.DataFrame.to_csv. This procedure not only reduces the number of features but also reduce the modelling cost, and in some case, improves the model performance. There are two limitations in creating the selected dataset. Only positively correlated features can be selected and the order of selection cannot be interchanged, i.e., selection is only possible in the given order as seen in Figure 10. For the fatigue dataset, there are 12 positive features, and the selected dataset can be written within this 12 features. As already specified the order of selection cannot be changed. One can select all the 12 or the first three/six. The end limit for selection is based on user input, referred as *selected_features* in click with a default end limit value of 12 and the same can be observed in fifth row of Table 5. If in case the limit input is greater than the number of positive correlations, only the positive correlated features will be written. These written datasets are used as an additional data to analyse the implemented predictive models.

3.5.3 Observations from CMatrix

From the correlation based feature ranking of fatigue dataset as seen in Figure 10, the fallowing observations can be made. Certain features, namely, Temparing time(Tt), Carburization Temperature(CT) and Diffusion Temperature(DT) have highest correlation(0.86 - 0.85) with the fatigue strength. Where as, Through Hardening Temperature(THT), Through Hardening Time(THt) and Cooling Rate for Tempering(TCr) have highest negative correlation(-0.66 and -0.65) with the fatigue strength. In case of a negative correlations, the increase of one feature will decreases the other and vice versa. The interpretation of the correlations are based on degree of correlation specified above. The dataset created in the feature selection method are used to analyse the impact of selected number of features with respect to the time taken by an regression model to execute and the accuracy of the model. The correlation matrix can also be created for feature selected dataset and extracted dataset, the same is depicted in Figure 12. The correlation of the extracted dataset is computed to verify, whether the principal components are linearly uncorrelated to each other as seen in Figure 12(b). This feature selected datasets can reduce the time taken by an predictive model to train with a little compromise in the model accuracy.

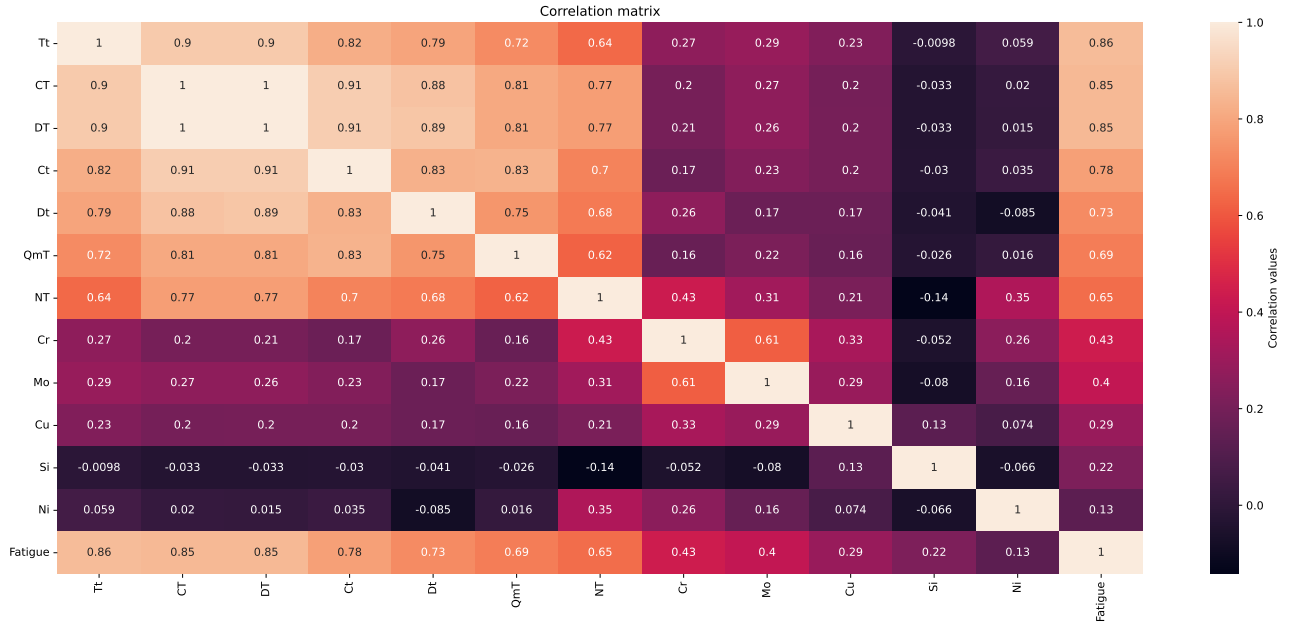


Figure 11: The Correlation matrix of 12 features that have positive correlation with the fatigue strength. The abbreviation for the features in x and y axis shall be referred from Table 1.

Generalization of CMatrix and other details:

- The CMatrix is design to work for any given dataset that complies the mentioned set of conditions.
- As a part of generalization **click** library is used as seen in Table 5 to adjust certain user defined inputs. These inputs can be given in the command line while executing the program, if not given, default values are set for every input and same can be seen in Table 5. To get more information on input options, try *filename.py - -help*.
- Each time when a plot is created or a file is written the program automatically creates a directory if one does not exist and store the results and the same can be found in the last two rows of Table 3.

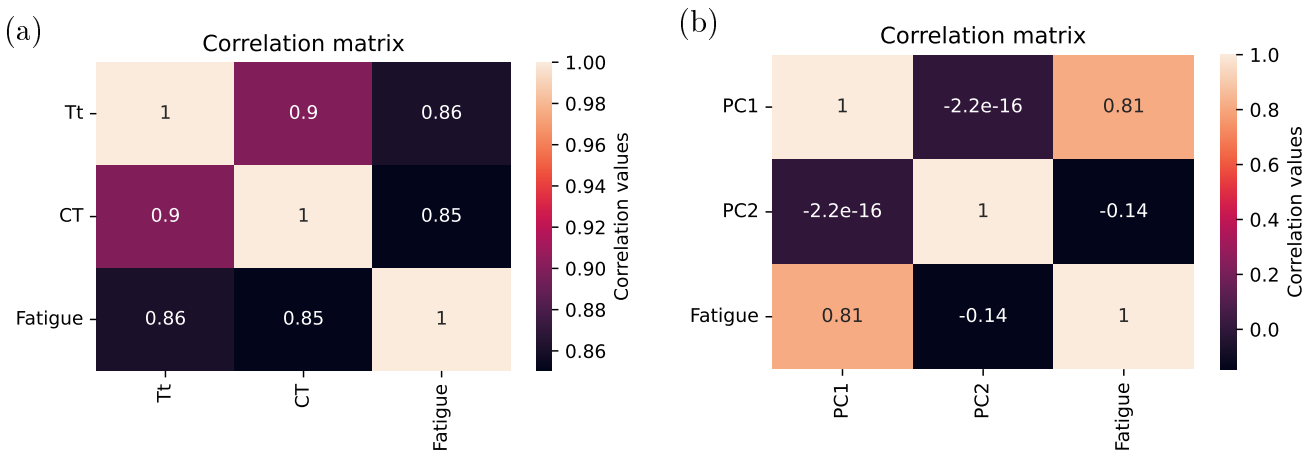


Figure 12: (a)Correlation matrix based on selected dataset with *selected_features* two; (b)Correlation matrix of extracted features confirming PC1 and PC2 are linearly uncorrelated.

- The program must be executed from the driver file *PPP_CMatrix_main.py* as all created features are already imported from the library *PPP_CMatrix.py* and the input datasets must be present in the same directory as the programming files.
- This task is additional to what was proposed in the data assessment section of the proposal.

Table 5: Python implementation of click in CMatrix with default values and the required input data types

click.option	nargs	dtype	default	help
- -data	1	str	fatigue_dataset.csv	Enter input dataset .csv: last column must be the target feature
- -target_column	1	str	Fatigue	Enter Target column header from dataset
- -targetf	1	str	Fatigue strength	Enter Target feature(for the ylabel of Figure 10)
- -yticks	3	float	[-1, 1.3, 0.2]	Enter yticks for Figure 10
- -selected_features	1	int	12	Select from Figure 10 the number of features to write
- -plt_size	4	float	[22, 9, 9.5, 7]	Enter plot size for Figure 9 and Figure 10 based on number of features

3.6 Multi Linear Regression

The Multi Linear Regression(MLR) is the first regression model that is implemented to predict the fatigue strength of steel. The MLR is a statistical technique that uses several independent features to predict the dependent feature as seen in Equation 4[7]. To compute the MLR model by default Gradient Descent(GD) optimizer is used where only the derivative of the loss function is consider during parameters update. The method implemented in this project is more robust and have adapted the program design of a neural network and uses Stochastic Gradient

Descent(SGD) optimizer. Here, the derivative of loss function, weights and biases are considered for parameters update as seen in Equation 10 and Listing 7. The MLR implementation is programmed into four different files with first one acting as a library implemented with required functions. The second one is the driver which can train and test the model, runs as a pure predictor, writes and plots training and testing results for single combination of loss(MSE/RMSE) and SGD optimizer. The third one is create-overall-results which creates results for two loss combinations with customizations for any given input data that full fills the dataset conditions. The last one is 'combination' which creates result analysis plot, writes predicted results to an excel and also writes time taken at different instances for the loss combinations. The functionalities implemented in the library will be discussed in the upcoming sections. The details of the features implemented in the MLR model can be found in Table 6.

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_n x_{in} + \epsilon \tag{4}$$

Here, y_i is the predicted feature of the i^{th} sample, β can be interpreted as weights, x_{in} is the n^{th} independent feature of the i^{th} sample and ϵ can be interpreted as bias.

Conditions to be considered before executing the files:

- *PPP_MLR_combination.py* should be executed only after executing *PPP_MLR_createOverallresults.py* at *predictor* : OFF(which is by default)
- In *PPP_MLR_combination.py* the input values should be same as the values used while executing *PPP_MLR_createOverallresults.py*

Table 6: Functionalities implemented in the MLR model

Functionality Implemented	Remarks	Functionality Implemented	Remarks
Import dataset	File format: .csv	SGD Optimizer	Hyperparameters: Learning rate and learning rate decay
Feature scaling	Standardizing features	Model training	Forward and backward propagation Training results
Test-train split	80% training set, 20% testing set	Store trained parameters	For predictor; File format: .npy and .txt only for visualizing
Single layer	Weights-bias initialization Random weights with reduced magnitude	Model prediction	Read trained parameters Prediction results
Updated layer	Trained parameters Used when executed as Predictor	Prediction comparision	Creates Excel to compare predicted vs target for two combinations
Loss functions	MSE and RMSE	Create results	Create results of two combinations
Coefficient of determination	$R^2 = 1$, for best case fit	Time taken	Time taken at different instance of the model

3.6.1 MLR Forward and Backward Propagation

The forward and backward propagation are the two important steps that are used in training the model. In the forward propagation step the MLR model predicts the fatigue strength and computes loss and accuracy of the model with respect to the predicted results and target feature. In the backward propagation step chain rule is applied to compute the derivative of loss

function, weights and biases. These parameter derivatives are then used in the optimization step to update the initially assigned parameters. To predict the output and to compute the derivative of the parameters a single layer function is implemented. This function is also called as the output layer. To compute the loss and its derivative, two different loss functions are implemented. These loss function refers to Mean squared error and Root mean squared error loss. To calculate the model accuracy, Coefficient of determination is implemented and for optimization step Stochastic gradient descent optimizer is used. The details on the working of each functions and their program design are provided in the upcoming sections.

3.6.2 Single layer

Single layer class of Multi Linear Regression(MLR) model uses tunable weights and biases on the independent features to predict the dependent feature. This single layer class consist of a constructor, a forward propagation(FP) step, a backward propagation(BP) step and read weights and biases function, which is used to read in the updated weights and biases after training the model. The weights and biases are initialized in the constructor. The size of the weights depends on the number of independent features(row) and the number of entries in the output layer(column), which is one as we are predicting for single output. The size of the bias will be [1,output layer]. The number of independent features will be assigned automatically with respect to the input dataset. The weights are randomly generated values based on numpy.random.seed(0) and biases are assigned as zeros. To reduce the magnitude of the generated Gaussian distribution the weights are multiplied by 0.01, thus, reducing the time taken by the model while training. These assigned values update inside the model during the training phase and the trained parameters are then used during model prediction. In the FP step, independent features are given as inputs and are used to compute the output as seen in Listing 3(a) by taking dot product with weights and adding the biases value. In BP step, we compute the derivative of the FP output with respect to weights which returns the inputs, with respect to biases which returns sum of one and with respect to inputs which returns the weights and the same can be seen in Listing 3(b). As BP is a chain rule, the derivative of loss function is multiplied with the computed derivative as seen in Listing 3(b). These derivatives of weights and biases are then used in the optimizer during parameters update.

```
# Single layer Forward propagation and Backward propagation
def forward_prop(self,inputs): #----(a)
  self.output = np.dot(inputs, self.weights) + self.biases
#-------------------------------------------------------------------------
def back_prop(self,derivatives): #----(b)
  self.weights_derv = np.dot(self.inputs.T, derivatives)
  self.biases_derv = np.sum(derivatives, axis = 0, keepdims = True)
  self.inputs_derv = np.dot(derivatives, self.weights.T)
```

Listing 3: Python implementation of Single layer Forward(a) and Backward(b) propagation step

3.6.3 Mean Squared Error Loss

The main method to calculate error in the regression models is the Mean squared error(MSE) loss, which measures the variance of the residuals(how well a line fits an individual data point). In MSE loss as seen in Equation 5[8], we square the difference between the dependent features(y) and the predicted output(y_pred) and find the average of those squared values. Loss computation is designed in a way that it has a derived class(MSE) that computes a vector of loss per sample in FP step and a base class called *LossCalculation* which calculates the mean value of the output vector(loss per sample) i.e., the loss of the model. In the FP step as seen in Listing 4(a),

the predicted output and dependent feature are given as inputs and are then substituted in Equation 5 to get loss of each sample as a vector. The partial derivative of the squared error with respect to predicted output is given by Equation 6 and the same is used in Listing 4(b) to compute the derivative. In the implementation, the FP output of the single layer(y_pred) and dependent feature(y) are used as inputs in FP step of MSE loss as seen in Listing 4(a), and the computed derivative is used as an input in the BP step of single layer as seen in Listing 3(b).

$$L_i = \frac{1}{J}\sum_j (y_{i,j} - \hat{y}_{i,j})^2 \tag{5}$$

$$\frac{\partial}{\partial \hat{y}_{i,j}} L_i = -\frac{2}{J}(y_{i,j} - \hat{y}_{i,j}) \tag{6}$$

Here, y is the dependent feature, $\hat{y}$ is the predicted output, i refers to the current sample, j refers to the current output of this sample and J refers to the number of outputs per sample.

```
# MSE loss Forward and Backward propagation
def forward_prop(self,y_pred, y): #----(a)
  lossPer_sample = np.mean((y - y_pred)**2, axis = -1)
  return lossPer_sample
#-------------------------------------------------------------------------
def back_prop(self, derivatives, y): #----(b)
  n_samples = len(derivatives) # Number of samples(rows)
  outputsPerSample = len(derivatives[0]) # Which is 1 w.r.t output sample
  self.inputs_derv = -2 * (y - derivatives) / outputsPerSample
  self.inputs_derv = self. inputs_derv / n_samples
```

Listing 4: Python implementation of MSE loss Forward(a) and Backward(b) propagation step

3.6.4 Root Mean Squared Error Loss

The Root mean squared error(RMSE) loss measures the standard deviation of residuals and it is defined as the square root of the MSE loss as seen in the Equation 7[8]. Here, RMSE loss will act as a derived class from which loss per each sample is computed and returned as a vector. The implementation and the function is same as the MSE loss as seen in Listing 5 with only changes in the equations of both FP and BP step. The partial derivative of the RMSE with respect to predicted output is given in Equation 8 and the same is used in BP step to compute the derivative. There also exist other loss computing metric like Mean Absolute error(MAE) loss which is more robust to data outliers(i.e., the observation that lies at an abnormal distance from other values) and measures the average of the residuals in the dataset.

$$L_i = \sqrt{\frac{1}{J}\sum_j (y_{i,j} - \hat{y}_{i,j})^2} \tag{7}$$

$$\frac{\partial}{\partial \hat{y}_{i,j}} L_i = \frac{\frac{-1}{J}(y_{i,j} - \hat{y}_{i,j})}{\sqrt{\frac{1}{J}(y_{i,j} - \hat{y}_{i,j})^2}} \tag{8}$$

Here, y is the dependent feature, $\hat{y}$ is the predicted output, i refers to the current sample, j refers to the current output of this sample and J refers to the number of outputs per sample.

```
# RMSE loss Forward and Backward propagation
def forward_prop(self,y_pred, y): #----(a)
  lossPer_sample = np.sqrt(np.mean((y - y_pred)**2, axis = -1))
  return lossPer_sample
#-------------------------------------------------------------------------
```

```
def back_prop(self, derivatives, y): #----(b)
  n_samples = len(derivatives) # Number of samples(rows)
  outputsPerSample = len(derivatives[0]) # Which is 1 w.r.t output sample
  self.inputs_derv = (-1 * (y - derivatives) / outputsPerSample)
   / np.sqrt((y - derivatives)**2 / outputsPerSample)
  self.inputs_derv = self. inputs_derv / n_samples
```

Listing 5: Python implementation of RMSE loss Forward(a) and Backward(b) propagation step

3.6.5 Coefficient of Determination

Coefficient of Determination(R^2) is defined as the proportion of the variation in the dependent feature(y) that is predictable from the independent features. The R^2 provides a measure of how well the observed dependent features are replicated by the model and results a value in the range from 0 to 1. It is computed as specified in Equation 9[8], in which the SS_{res} refers to residual sum of squares and SS_{tot} refers to the total sum of squares. The best possible result is when the predicted values exactly matches the value of the target feature, which results in $SS_{res} = 0$ and $R^2 = 1$. When the model makes a worst prediction then the resultant R^2 will be a negative value. The same equations are implemented in the function that computes R^2 with predicted output(y_pred) and dependent features(y) as input and the same can be seen in Listing 6.

$$\begin{aligned} SS_{res} &= \sum_i (y_i - \hat{y}_i)^2 \\ SS_{tot} &= \sum_i (y_i - \bar{y})^2 \\ R^2 &= 1 - \frac{SS_{res}}{SS_{tot}} \end{aligned} \tag{9}$$

Here, y is the dependent feature, $\hat{y}$ is the predicted output and $\bar{y}$ is the mean value of the dependent feature.

```
# Computing accuracy of the model
def coefficient_of_determination(self,y_pred,y):
  SSres = np.sum((y - y_pred)**2) # Sum of squares of residuals
  SStot = np.sum((y- np.mean(y))**2) # Total sum of squares
  Rsqr = 1 - (SSres/SStot)
  return Rsqr
```

Listing 6: Python implementation to compute Coefficient of Determination

3.6.6 Stochastic Gradient Descent Optimizer

Once the back propagation step is done, the computed derivatives are used to adjust the weights and biases to decrease the measured model loss. This adjustments are done using optimizers, and here, Stochastic gradient descent(SGD) optimizer is implemented. There also exist other optimizers which will be discussed in the coming sections. In the SGD optimizer as seen in Equation 10[9], the parameters are updated during each training step with high variance that causes the objective function to fluctuate heavily. The SGD optimizer consist of two tunable hyperparameters, learning rate and learning rate decay. Learning rate is an important parameter as it is responsible for the weightage of parameters-update that occurs during optimization step as seen in Listing 7(b). If not tuned properly, and if the values are high, then the model gets stuck in local minima i.e., the loss value does not change for the remaining number of epochs and results in a trained model that is not suitable for prediction. Fixing a learning rate through out the training step is not an ideal solution as it may find

another local minimum. To avoid this, we have to decrease the learning rate over each number of epoch. This is achieved by introducing a new hyperparameter called learning rate decay or exponential decay. The learning rate decay, decreases the learning rate for each training step and creates a current learning rate($C\_learning\_R$), which is then used to update the parameters and same is implemented in Listing 7(a). The results of the learning rate decay can be referred from the file $trainingresults\_MSE/RMSE.txt$ present in the directory Results_-Training. For the implemented MLR model, with respect to the given dataset, the hyper parameters are tuned and the respective values as shown in Table 7 are selected for learning rate and learning decay, as they produce lower loss and higher accuracy. These values are set by default in click implementation and are tunable for other input datasets. In the implementation part, an $SGD\_Optimizer$ class is created and it consist of a constructor that initializes the input hyperparameters and two important function, $learning\_R\_update$ and $parameters\_-update$. Among the two, the former creates current learning rate as discussed before and can be seen in Listing 7(a). The later, updates the parameter for each epoch by accessing the layer, which is the input of the function $parameters\_update$ and the same can be seen in Listing 7(b).

$$\theta = \theta - \eta.\nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{10}$$

Here, θ refers to the parameters such as weights and bias, η refers to the learning rate and $J(\theta)$ refers to the gradient of the parameters calculated during back propagation step.

Table 7: SGD optimizer tuned hyperparameters at MSE and RMSE loss for MLR and ANN models

Combinations	Learning rate	Learning rate decay
MLR_MSE	0.13	1e-1
MLR_RMSE	0.5	1e-1
ANN_MSE	0.85	1e-1
ANN_RMSE	0.85	1e-2

```
# SGD optimizer learning rate update and parameter update
def learning_R_update(self): #----(a)
  self.C_learning_R = self.learning_R * (1. / (1. +self.learning_R_decay *
   self.itr))
    return self.C_learning_R
#-------------------------------------------------------------------------
def parameters_update(self, layer): #----(b)
  layer.weights += -self.C_learning_R * layer.weights_derv
  layer.biases += -self.C_learning_R * layer.biases_derv
```

Listing 7: Python implementation of SGD optimizer learning rate update(a) and parameters update(b)

3.6.7 MLR Data preprocessing

The first step in the MLR implementation is the data preprocessing step which is done to transform raw input data into a readable and understandable format. The $Data\_preprocessing$ class consists of three functions, namely, $import\_dataset$, $feature\_scaling$ and $split\_dataset$. Here, the import dataset and scaling function are same as what was discussed in K-means. But, the $feature\_scaling$ returns additional two values i.e., the mean and the standard derivation of

the dependent feature which are then used to re-scale the predicted results. In addition, the input dataset must also comply to the conditions that are specified in K-means. In MLR model we perform both training and testing step and thus requires training and testing set of the scaled independent and dependent features. This is achieved by using the function *split_dataset* which randomly splits the scaled features into training set and testing set, i.e., 80% of the samples are for training set and remaining 20% is for the testing test. For example, the size of the fatigue dataset is [437 x 25], after splitting, the size of the training set at independent feature is [349,25] and testing set is [88,25]. The randomness of the split is achieved by using pd.DataFrame.sample at $random\_state = 0$. In MLR model the first independent feature(Column) also called as y-intercept must be filled with ones. Thus, the training(X_train) and testing set(X_test) of the independent features(X) will have ones in the first column i.e., $X\_train.shape$ will be [349,26] because of the additional column and this is done using numpy(np).hstack. These updated sets will then be used during model training and prediction.

3.6.8 MLR model Training

Training is an important step in model construction as it is responsible for creating ideal weights and biases for predicting the target feature. The updated training set i.e., X_train and y_train from the data preprocessing step will be used as the independent features and testing metric during training. In training, the model repeatedly perform a forward pass, backward pass and optimization, until it reach a stopping point which is controlled by the number of epoch +1. Here, +1 represents one more FP step with the latest updated parameters from the final epoch. Each full pass through all of the training data is called an epoch and it is a hyper parameter that should be greater than zero, if not, the program exits with an error message: "Error: no_of_epoch input must be > 0". By default the number of epoch assigned for training the model is 10,000 and it can be tuned based on the dataset. For training the model, loss metric must be selected and it is done based on user input. In click, there are two options for the user to enter, it can be 'MSE' or 'RMSE', and when no input is given MSE will be the default option, and if some other options are given, the program will exit with an error message: "Error: Recheck *loss* input; Possible inputs MSE or RMSE". But, when we are creating the overall results the model will be trained for both losses. The training results of the model for every 100^{th} epoch is written using pd.DataFrame.to_markdown and saved into a file $trainingresults\_MSE/RMSE.txt$. The file also consists of abbreviation details and to tabulate the results in such grid format as seen in Figure 14, one should make sure that 'tabulate' package is already installed. In addition, the resultant training plot for the tuned hyperparameters depicted in Figure 13 contains the results of four function, namely, training results, loss convergence, accuracy convergence and learning rate decay. From this plot, one can identify whether a model has trained or stuck in the local minima. After training, the updated parameters are written into a .npy or .txt file and the file format is selected based on user input. The default option is .npy, as the read-in function looks for this format and .txt is only for visualising the parameters. If other format are given as input, the program will exit with an error message: "Error: Recheck $writeparam\_as$ input; Possible inputs .npy or .txt". All these above mentioned checking procedures are executed before any of the process begins.

3.6.9 MLR model Prediction

The prediction step is to validate the trained model whether it is good enough to make prediction with higher accuracy rate and lower loss for unseen datasets that are similar to the ones that was used for training the model. If the resulting accuracy and loss values are not closer to ones that are in training results, then our model is said to be over-fitting. This means, the model is just memorizing the data without any understanding of the dataset. An over-fit

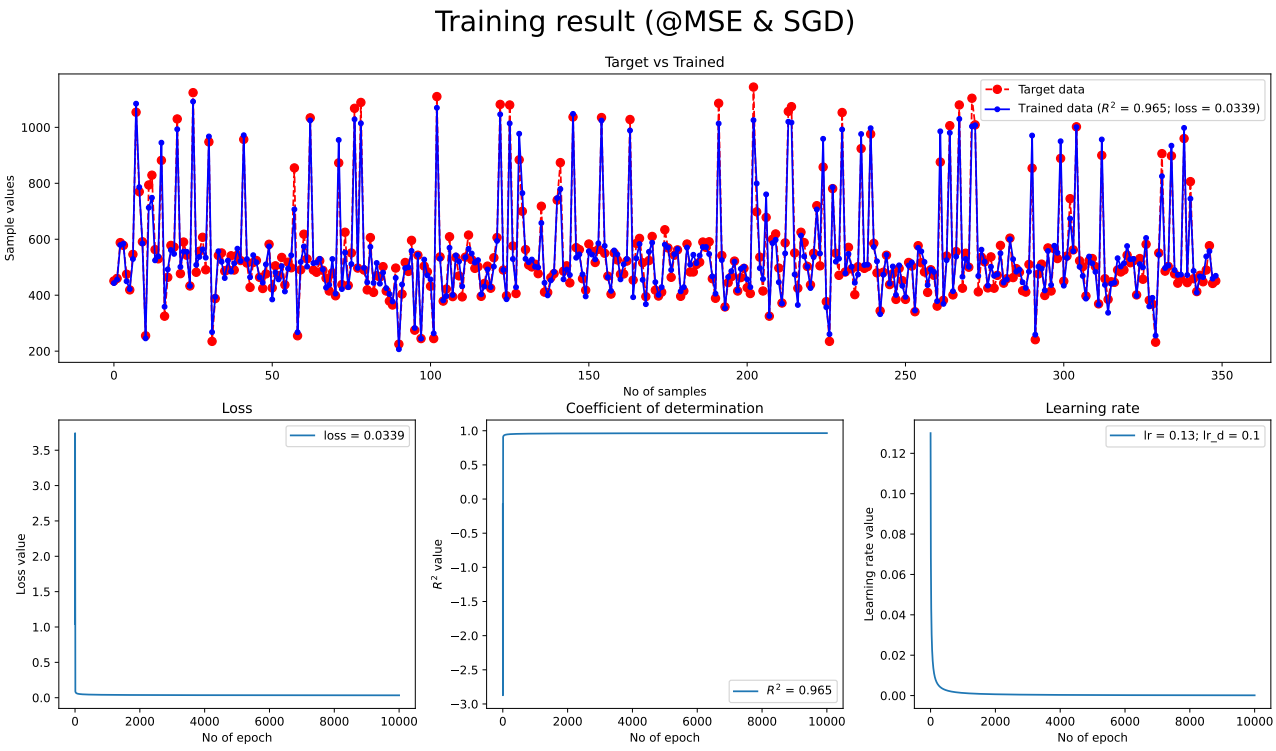


Figure 13: MLR training results plot for fatigue dataset at Loss metric: MSE.

Training results

Loss and optimizer used: MSE, SGD

Epoch	R^2	Loss	lR
0	-0.0787145	1.04203	0.13
100	0.943398	0.0546775	0.0118182
200	0.949213	0.0490603	0.00619048
300	0.951967	0.0463993	0.00419355
400	0.953686	0.0447388	0.00317073

Figure 14: The first five entries of the MLR training results written at every 100th epoch for fatigue dataset at Loss metric: MSE.

model will do very well predicting the data that it has already seen, but often significantly worse on unseen data. In this project the updated testing set i.e., X_test and y_test from the data preprocessing step will be used as the unseen data and the testing metric during prediction. Also to validate our implementation we can use the training set(X_train) instead of test set(X_test). This means using the same dataset that is used for training the model and checking whether our model over-fits the data with 100% training accuracy. In model prediction step a single forward propagation is done for the selected dataset. Here, there are two options in click for dataset selection i.e., 'testset' to check the models performance or 'trainingset' to check the model implementation. By default, testset is selected and if any other input is given the program will exit with an error message: "Error: Recheck $pred\_dataset$ input; Possible inputs testset or trainingset". After the prediction the results are

written into a file *testingresults_MSE/RMSE.txt* and the same can be seen in Figure 16. The prediction plot plotted with respect to the predicted results and target results is saved as *plot_testingresults_MSE/RMSE.png* and same is depicted in Figure 15(a). In addition to this, a prediction error plot is plotted as seen in Figure 15(b), by computing the error between the target results and the predicted ones. The predicted results and the target results are then re-scaled using the mean and standard deviation computed from the *feature_scaling* function to compare with each other. To the check the correct of the rescaling step, the accuracy of the results before rescaling and after rescaling is compared, and the results of the same along with the time taken results are saved into a file *resultcomparision_MSE/RMSE.txt*.

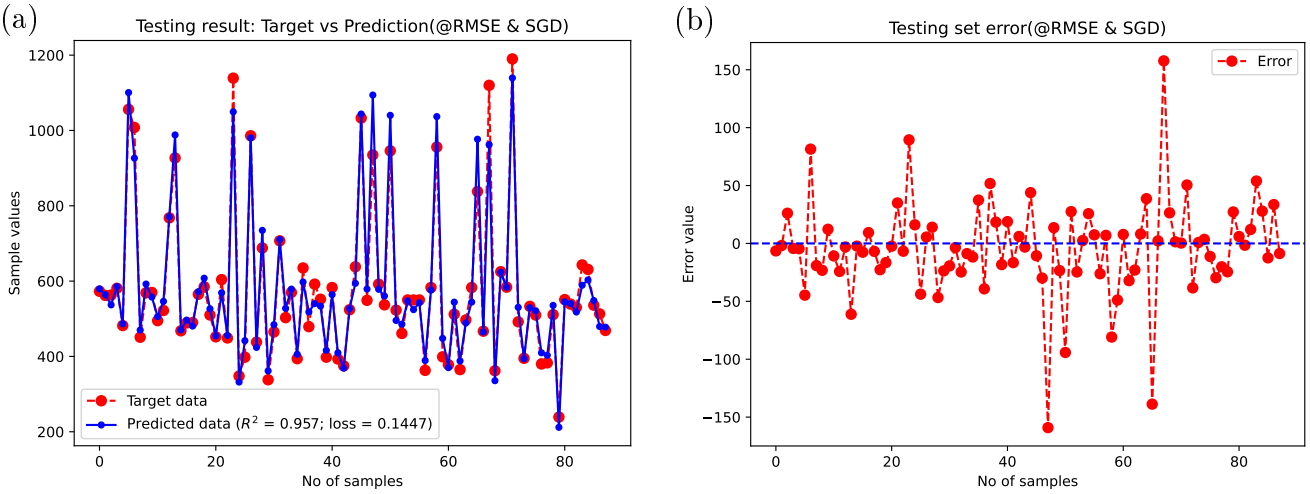


Figure 15: (a)MLR testing results and (b)testing error plot for fatigue dataset at Loss metric: RMSE.

```
Testing results

Loss and optimizer used: MSE, SGD

testing:  R^2: 0.9591319979102966, loss: 0.04551727363485865

Abbreviations:
R^2: Coefficient of determination(Accuracy)
```

Figure 16: MLR testing results for fatigue dataset at Loss metric: MSE.

3.6.10 MLR model as pure Predictor

After the prediction step, if the model has fulfilled the user criteria with expected accuracy, without any over fitting, and if the user wants to predict for more samples, then the user can execute the program as a pure predictor. Here, more samples refers to new datasets that are similar to or an extension of the dataset that was used during model training. The idea behind this procedure is to avoid re-training of the model and read-in the stored updated parameters to predict results. Here, there is no need for splitting the input dataset. The scaled dataset(*scaled_X* from feature scaling) with first column inserted with ones is directly considered for prediction and *scaled_y* will be considered as the test metric. The implemented single layer function can not be used for pure predictor as weights and biases are already assigned. A new function called Updated layer is created. This function uses the trained

weights and biases to perform FP step. This predictor is implemented in the prediction step and will be switched "ON" or "OFF" based on user input. By default it will be OFF and the program will be executed normally. If other option are given as input, the program will exit with an error message: "Error: Recheck *predictor* input; Possible inputs ON or OFF". The results created during the pure predictor step are testing results and results comparison. The file names of these results are same as the ones in prediction step but these results are stored in a different directory **Results_MLR_predictor**. Also, the time taken for the model to execute and to predict is written into the results comparison file.

Conditions to be considered before executing pure predictor:

- The model should be executed as a predictor only after training the model with dataset similar to that of the predictor dataset.
- The loss used during predictor should be same as the loss used while training the model.
- When the *predictor* is ON no training will be done and thus no training results will be created.
- If overall results are created with *predictor*: ON, then only predictor results are created which is not sufficient to execute the combination code $PPP\_MLR\_combination.py$.

3.6.11 Role of creator and combination code

Create overall results: The $PPP\_MLR\_createOverallresults.py$ is used to create results in a single go for a given input dataset in both combinations of loss functions and two different hyperparameters set for SGD optimizer as seen in Table 7. This operation is done using the **subprocess** library which executes the driver code $PPP\_MLR\_main.py$ for two different loss functions with customisable inputs through click. The major advantage of this code is it makes hyperparameter tuning and validation of the MLR model for a given dataset easier rather than tuning each combination separately for different cases. It can also execute as a pure predictor using the trained parameters computed at both loss functions and notes the time taken by each loss function and both loss function to execute. The options of create overall results for a given input dataset can be referred from Table 9 with respect to the column program used.

Combination: The $PPP\_MLR\_combination.py$ program should be used after $PPP\_MLR\_createOverallresults.py$ to create overall results plot as depicted in Figure 31, time taken results as seen in Table 27 and comparison excel for a given input dataset. The highlighted files and plots present in Table 8 are ones that were created during this process. The comparison excel consist of three columns, one with test metric(i.e., the target results) and other two with prediction results of both combinations. In the end of the file the mean and standard deviation values of each columns are also provided. This .xlsx file is written using pd.ExcelWriter and pd.DataFrame.to_excel and saved as $resultcomparisionMLR.xlsx$. The purpose of the file is to compare the target results of the fatigue strength for each sample with results obtained by the model during prediction and find out which combination has made better prediction for a particular sample. For example, the first sample of the file $resultcomparisionMLR.xlsx$ has a target value: 573; prediction at MSE: 553 and prediction at RMSE: 579. With respect to the observation we can say that for this sample RMSE has made better prediction than MSE. Likewise, we can do this for all the samples and find out which combination among the two has highest number of better prediction of the target feature. This file is also analogous to the error plot as seen in Figure 15(b). This comparison can be considered as an additional metric to select the better performing MLR model irrespective of the computed R^2 value.

Table 8: Files and directories created during MLR implementation and the highlighted ones are created during *PPP_MLR_combination.py*

Directory	Files and plots	Remarks
Plots_MLR_Main	plot_testingerror_MSE/RMSE.png plot_testingresults_MSE/RMSE.png plot_trainingresults_MSE/RMSE.png	Plots of Training and testing results and testing error as seen in Figure 13 and Figure 15(a)(b)
Plots_MLR_combination	plotOverall_results.png	Overall results plot as depicted in Figure 31
Results_Training	trainingresults_MSE/RMSE.txt	Tabulated training results at MSE and RMSE as seen in Figure 14
Results_Testing	testingresults_MSE/RMSE.txt	Testing results at MSE and RMSE as seen in Figure 16
Results_Parameters	parametersMLR_MSE/RMSE.npy/txt	Trained parameters
Results_MLR_predictor	testingresults_MSE/RMSE.txt resultcomparision_MSE/RMSE.txt	Testing results and results comparison table at MSE and RMSE created during pure predictor
Results_TargetVSpred	resultcomparison_MSE/RMSE.txt resultcomparisonMLR.xlsx timetaken_overall.txt	Result comparison table at MSE and RMSE, overall results comparison Excel and time taken in create overall results
Results_Timetaken	resulttimetakenMLR.txt	Time taken results of the model for a given dataset as seen in Table 27

3.6.12 Generalization of MLR model and other details

Table 9: Python implementation of click in MLR with default values and the required input data types for all the three programs except the library

click.option	nargs	dtype	default	help	Program used
- -data	1	str	fatigue_dataset .csv	Enter input dataset .csv: last column must be the target feature	All the three
- -no_of_epoch	1	int	10001	Enter No of epoch for training(>0)	driver and overall
- -loss	1	str	MSE	Select loss: [MSE or RMSE]	driver and combination
- -losslist	2	str	[MSE, RMSE]	Loss input as a list	Overall and combination
- -sgd	2	float	[0.13, 1e-1]	Enter SGD_optimizer input	driver and overall
- -sgd_rmse	2	float	[0.5, 1e-1]	Enter SGD_optimizer input @RMSE	Overall
- -pred_dataset	1	str	testset	Select dataset for prediction: [testset or trainingset]	driver and overall
- -predictor	1	str	OFF	Select ON or OFF for only prediction	driver and overall
- -makeplot	1	int	1	Select 0 or 1 to makeplots for single combination	driver and overall
- -writeparam_as	1	str	npy	Select write format: npy or txt	driver and overall
- -targetf	1	str	Fatigue Strength	Enter target feature for Figure 31	Combination
- -limacc	2	float	[0.95, 0.97]	Enter lower and upper bound for acceleration values of Figure 31	Combination
- -limls	2	float	[0.02, 0.15]	Enter lower and upper bound @MSE & RMSE values for Figure 31	Combination

- The implemented MLR model is design to work for any given dataset that complies the mentioned set of conditions for dataset.
- As a part of generalization **click** library is used as seen in Table 9 to adjust certain

user defined inputs. These inputs can be given in the command line while executing the program, if not given, default values are set for every input and same can be seen in Table 9. To get more information on input options, try *filename.py - -help*.

- Each time when a plot is created or a file is written the program automatically creates a directory if one does not exist and store the results and the same can be observed in the Table 8
- All created features in the library *PPP_MLR.py* are imported into the driver *PPP_MLR_main.py* and the input datasets must be present in the same directory as the programming files.
- This confirms the completion of the task Multi linear regression as specified in the proposal.

3.7 Artificial Neural Networks for Regression

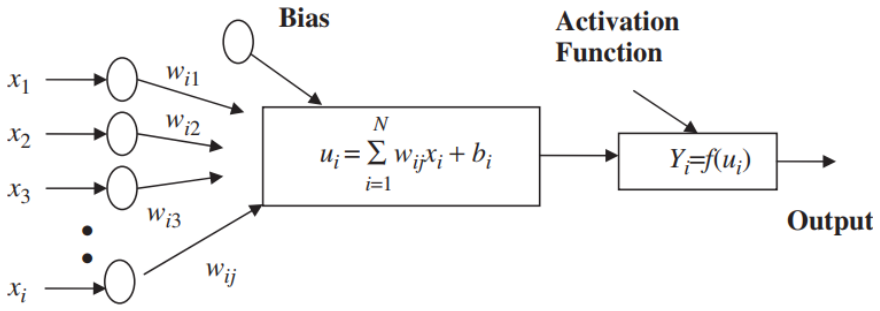


Figure 17: Basic elements of an artificial neuron[10].

The Artificial Neural Networks(ANNs) or in this case deep neural networks with three layers is the second regression model that is implemented to predict the fatigue strength of steel. The neural networks are a subset of machine learning and deep learning algorithms that are inspired by the human brain as it mimics the way biological neurons signal to one another. The basic element of an NN is the artificial neuron as shown in Figure 17 which consists of three main components, namely, weights, bias, and an activation function. Each neuron receives inputs $x_1, x_2, ..., x_n$ attached with a weight w_i which shows the connection strength for that input for each connection. Each input is then multiplied by the corresponding weight of the neuron connection. A bias b_i can be defined as a type of connection weight added to the summation of inputs and corresponding weights u_i is given by Equation 11. The summation u_i is transformed using a scalar-to-scalar function called an activation function Y_i, and the same is also given by Equation 11[10]. This results in a pattern in which the output of one layer being used as an input to the next layer. This type of network is called as the feed forward neural network and the same is used in this project. The model implemented consist of an input layer, two hidden layers and a output layer as seen in Figure 19. Two activation functions are used in this model, Rectified linear activation function(ReLU) for the hidden layers and linear activation function for the output layer. The ANN implemented is programmed into four different files. The first one acts as a library implemented with required functions and four different optimizers. The second one is the driver which can train and validate the model, run as a pure predictor, write data to plot and writes and plots training and testing results for single combination of loss(MSE/RMSE) and optimizer(SGD/SGDM/RMSP/Adam). The third one is to create-overall-results for all the eight combinations of loss and optimizers, and

to tune hyperparameters using adjustable inputs for any given dataset that full fills the dataset conditions. The final one is to make comparison plots of the hyperparameter tuning, creates result analysis plots, writes predicted results of eight combinations into an excel and writes time taken results at different instances for all the combinations. The functionalities implemented in the library will be discussed in the upcoming sections. In this project the program design of MLR was adapted from neural networks, thus, the program flow between the two will be similar and the structure of some of the implemented functions in MLR will also hold for ANN with some required modifications and additional details. The details of the features implemented in the ANN model can be found in Table 10.

$$u_i = \sum_{j=1}^{H} w_{ij} x_j + b_i$$
$$Y_i = f(u_I) \tag{11}$$

Table 10: Functionalities implemented in the ANN model

Functionality Implemented	Remarks	Functionality Implemented	Remarks
Import dataset	File format: .csv	Optimizers	SGD, SGDMomentum, RMSProp and Adam
Feature scaling	Standardizing features	Model training	Forward and backward propagation Training results
Test-train split	80% training set, 20% testing set	Hyperparameter tuning	HP of hidden layers, regularization loss and optimizers
Dense layer	3 layers - 2 Hidden and 1 output layer Weights-bias initialization, Random weights with reduced magnitude	Store datas to plot	For comparision plot; File format: .txt
Activation function	ReLU and Linear	Store trained parameters	For predictor; File format: .npy and .txt only for visualizing
Updated layer	Trained parameters Used when executed as Predictor	Model prediction	Read trained parameters Prediction results
Loss functions	MSE and RMSE	Prediction comparision	Creates Excel to compare predicted vs target for eight combinations
L2 Regularization	Regularization loss	Create results	Create results of eight combinations
Coefficient of determination	$R^2 = 1$, for best case fit	Time taken	Time taken at different instance of the model

3.7.1 ANN Forward and Backward Propagation

ANN forward and backward propagation step is a bit different from what was discussed in the MLR section(see Sect.3.6.1). Unlike in the MLR model where only a single layer was used (output layer), the ANN model has two hidden layers and an output layer, this calls for the implementation of dense layer. In the forward propagation(FP) of the ANN model the procedure shown in Figure 17 is repeated for each dense layer(1,2 and 3) and activation function. As it is a feed forward neural network, the output of the activation function will serve as the input for the next layer until the activation function of the output layer, which returns the predicted fatigue strength. This predicted results are then used to compute the total losses and accuracy of the model. In order to improve the models performance, backward propagation(BP) step is introduced. Here, the chain rule is applied to compute the derivatives of the weights and biases of each layer, which are then used in the optimization step to update

the parameters. Similar to forward propagation, where we propagated from the input layer until the computation of loss function. In backward propagation, we propagate backward by computing the derivative of the loss function, then the derivative of output layer activation function, the output layer itself, and likewise repeated until we compute the derivative of the first layer. From this procedure the parameters derivative of each dense layer is computed. These computed weights and biases are then used to update the existing parameters during the optimization step. To perform forward and backward propagation the fallowing functions are implemented. Dense layer class for layers, ReLU(for hidden layer) and linear(for output layer) activation class for activation function, MSE, RMSE and L2 regularization loss for loss computation and Coefficient of determination for accuracy. For the optimization step SGD, SGDMomentum, RMSProp and Adam optimizers are implemented. More details on each function and its working is provided in the upcoming sections.

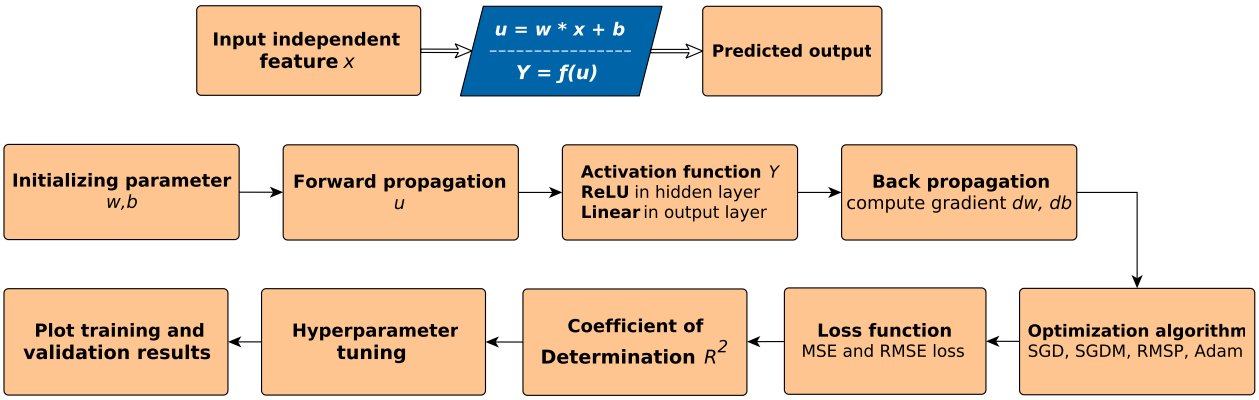


Figure 18: Work flow of the implemented neural network.

3.7.2 Dense layer

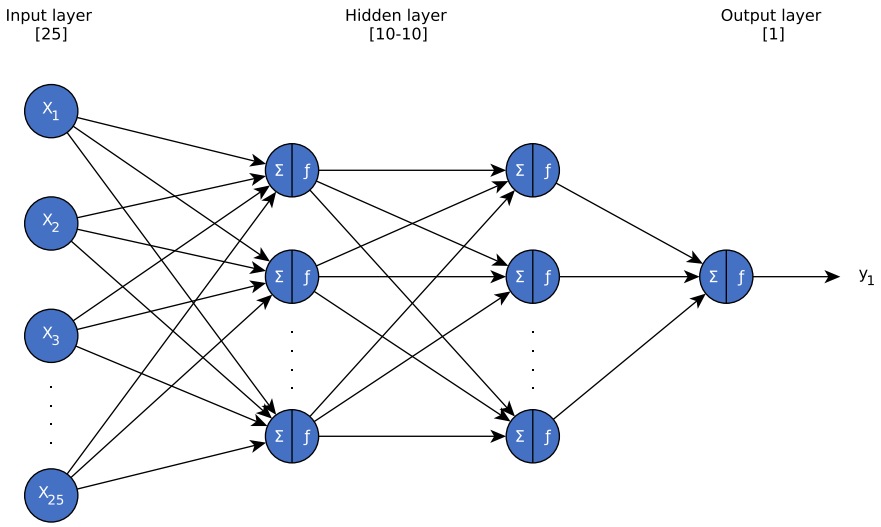


Figure 19: Implemented Neural Network architecture [25-10-10-1].

The algorithmic implementation and the working of the Dense layer class is adopted from the Single layer class(Sect.3.6.2) with only modifications in the backward propagation(BP) step. Here, along with the weights and biases the regularization loss is also initialized in the

constructor. As we know from the Sect.3.6.2, the weights are randomly generated values and the biases are zeros. Here, we require three layers, two for two hidden layers and one for the output layer. The number of neurons present in a hidden layer is selected based on hyperparameter tuning. In case of fatigue dataset, the model predicts with higher accuracy at 10 neurons for both hidden layers and the same can be seen from Figure 33.

Working of the Dense layer: For the first layer, the size of the weights depends on the number of independent features(row) present in the input dataset and the number of neurons required for the first hidden layer(column). The size of the biases are represented as [1,hidden layer]. The number of columns in a bias should always be equivalent to number of neurons in a layer. In the first FP step as seen in Listing 3(a), the computed output is the hidden layer with 10 neurons. From Figure 17, it can be seen that the output of a layer in this case hidden layer will be the input of the activation function, in this case it will be the ReLU activation. The size of the activation function output will be the same as its input, the number of neurons(10) in the output can also be referred as the independent features of the layer. As FP is a feed forward type, the output of activation will be the input for second layer. The size of the weights for this layer will be no of neurons in the activation output(row) and the number of neurons(10) required in the second hidden layer(column). The same procedure is then repeated and the output of ReLU will be the input for the third layer. For this layer, the size of the weight will be number of neurons from the output(row) and output layer(column) which is 1. The activation function used in the output layer is linear, and the output of the function will be the predicted results. The dense layer class should be called for each layer with number of independent features, number of neurons required in the hidden layer and lambda hyperparameters for regularizations as inputs. The number of independent features will be assigned automatically with respect to the functions input. The number of neurons and lambda in the hidden layers are tunable hyperparameters based on user inputs. The scope of the dense layer FP function is to compute the output which will serve as an input for activation function. The BP step of the dense layer is same as implemented in Sect.3.6.2, with only modification is the computed derivatives of the parameters are updated by the derivatives of the regularization loss for each layers and the same can be seen in Listing 8. This BP procedure is the major step in model training.

```
# Dense layer Backward propagation
def back_prop(self,derivatives):
  self.weights_derv = np.dot(self.inputs.T, derivatives)
  self.biases_derv = np.sum(derivatives, axis = 0, keepdims = True)
  # Regularization derivative at weights
    if self.L2_weight_reg > 0: # L2 Reg weight input
       self.weights_derv += 2 * self.L2_weight_reg * self.weights
    # Regularization derivative at biases
    if self.L2_bias_reg > 0:   # L2 Reg bias input
       self.biases_derv += 2 * self.L2_bias_reg * self.biases
  self.inputs_derv = np.dot(derivatives, self.weights.T)
```

Listing 8: Python implementation of Dense layer Backward propagation step

3.7.3 Activation function

An activation function in a neural network defines how the weighted sum of the input is transformed into an output from a neuron or neurons in a layer of the network. Many activation functions are non-linear and may be referred to as the non-linearity in the layer or the network design. The choice of activation function has a large impact on the capability and performance of the neural network, and different activation functions may be used in different parts of the

model. All hidden layers typically use the same activation function. The output layer will typically use a different activation function from the hidden layers and is dependent upon the type of prediction required by the model. Typically, a differentiable non-linear activation function is used in the hidden layers of a neural network. This allows the model to learn more complex functions than a network trained using a linear activation function [11]. In the implemented ANN, ReLU activation is used in the hidden layer and linear activation is used in the output layer as shown in Figure 20. There also exits other activation functions for hidden layer, like, logistic(sigmoid) which outputs in the range of 0 to 1 and Hyperbolic Tangent(Tanh) which is similar to sigmoid and outputs in the range of -1 to 1. The softmax activation for the output layer is used for classification problems. ReLU activation is preferred in the hidden layer because of its higher efficiency compared to sigmoid and tanh. In the FP step hidden layers are given as inputs. Here, the ReLU function returns zero if the input(x) is negative, otherwise, the value. In the BP step, the BP outputs of the dense layer will be the inputs. Here, the function returns one if input(x) is greater than zero, otherwise, returns zero. Linear activation also called as no activation is preferred in the output layer as it does not change the weighted sum of the input and passes it to the output. The BP of linear activation return one and same can be seen from Figure 20.

Activation function	Forward propagation	Backward propagation(Derivative)
ReLU	$f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}$
Linear	$y = x$	1

Figure 20: Forward and backward propagation equations of ReLU and Linear Activation functions.

3.7.4 Loss calculation

After computing the prediction results the model loss is calculated with respect to the target feature. In the implemented ANN model both MSE and RMSE loss are used to compute the model loss. These loss metrics are extensively discussed in the Sect.3.6.3 and 3.6.4 and the same holds for the ANN model.

L2 Regularization loss: In addition to the loss functions, L2 regularization technique is introduced to avoid overfitting issues by adding extra information to the model and improve models performance. This L2 regularization is also called as the Ridge regression which is a type of linear regression that introduces a small amount of bias, know as Ridge regression penalty to get better predictions. L2 regularization as given in Equation 12[12], computes the sum of the squared weights and biases of the implemented layers. This type of regularization drives sum of the parameters towards zero and are used mostly in the hidden layers. The lambda in Equation 12 is the tunable strength hyperparameter which is given as an input in the dense layer class. In the forward propagation step, dense layers are given as input, from which the lambda parameters, weights and biases of each layers are used to compute the L2 loss. The computed loss values of each layer is then summed together to form regularization loss. The resultant loss is then added to the loss function which results in the total loss of the model. In the back propagation step as seen in Listing 8 the derivative of the regularization with respect

to weights are added to the weights derivative and the same for biases in each hidden layer.

$$L_{2w} = \lambda \sum_{i=1}^{n} w_i^2$$
$$L_{2b} = \lambda \sum_{j=1}^{n} b_j^2 \tag{12}$$

Here, L_{2w} and L_{2b} are the regularization at weights and biases, i refers to the iterator that iterates over all the weights in a layer and j is the iterator for biases.

3.7.5 Optimizers

To decrease the ANN model loss, the weights and biases in each layer is optimized using the derivatives computed during the back propagation step. This adjustment or update of the originally assigned weights and biases are done using optimizers. In ANN, four different optimizers are used, namely, Stochastic gradient descent(SGD), Stochastic gradient descent with momentum(SGDM), Root mean square propagation Optimizer(RMSP) and Adaptive momentum Optimizer(Adam). All four implemented optimizers are programmed in the same way with only changes required during the parameters update step. The SGD optimizer is discussed in detail in Sect.3.6.6, and other three optimizers will be discussed in the upcoming sections.

3.7.6 Stochastic gradient descent with momentum Optimizer

The SGDMomentum(SGDM) optimizer as seen in Equation 13[9] is the modified version of the SGD optimizer that helps to accelerate the gradients vectors in the right direction and leads to faster convergence. Consider a ball rolling down the hill, for each epoch the ball will gain momentum and rolls faster towards the local minima. This reaches a better local minima than in SGD optimizer and further decreasing the model loss. In the momentum optimizer as seen in Equation 13, we set a momentum parameter(γ) that retains a fraction of the previous parameter(v_{t-1}), and subtracts the computed gradient($\nabla_\theta J(\theta)$) multiplied by the learning rate(η), from it. The resultant is the parameter update(v_t), which consist of the portion of gradient from the previous steps as our momentum and the portion of the computed gradient. The same is implemented in the parameters update step, which updates the weights and biases of each layer, for each epoch during model training. The SGDM optimizer consist of three tunable hyperparameters, learning rate, learning rate decay and momentum. The importance of learning rate and its decay is discussed in detail in Sect.3.6.6. In case of momentum, if the fraction is set too high, the model might stop learning as the direction of updates will not fallow the global gradient decent. For the implemented ANN model, with respect to the fatigue dataset, the hyperparameters are tuned as seen in Figure 32, and the respective values as shown in Table 11 are selected. These values are selected as they produce lower prediction loss and higher model accuracy. The implementation is same as specified in Listing 7, with modifications in parameters update step with respect to Equation 13.

$$v_t = \gamma v_{t-1} - \eta . \nabla_\theta J(\theta)$$
$$\theta = \theta + v_t \tag{13}$$

Here, v_t refers to the parameters update, γ is the momentum co-efficient, v_{t-1} is the parameters update of the previous step, η is the learning rate, θ refers to the parameters such as weights and bias, and $J(\theta)$ refers to the gradient of the parameters calculated during back propagation step.

Table 11: SGDMomentum optimizer tuned hyperparameters at MSE and RMSE loss for ANN model

Combinations	Learning rate	Learning rate decay	Momentum
MSE and RMSE	0.85	1e-1	0.6

3.7.7 Root mean square propagation Optimizer

The RMSProp(RMSP) optimizer as seen in Equation 14[9] is similar to SGDMomentum optimizer with relatively faster convergence rate. In addition to momentum, the mechanism used in RMSProp also has a per-parameter adaptive learning rate which makes the learning rate changes smoother. This results in slower learning rate change with benefit of fast convergence. An important property of RMSP is that, for each update of the squared gradient($E[g^2]_t$), not only a part of the squared gradient is retained(using ρ), but also updated with the fraction($1-\rho$) of new squared gradient. A new hyperparameter ρ, is considered as a moving average parameter or a memory decay rate. This is due to the fact that, the default values has so much momentum of gradient and learning rate updates, even a smaller gradient update will allow the optimizer to perform better. In RMSP, we have to be careful with the learning rate input as the model gets unstable if a large value is selected. In the implementation, the model is trained with RMSProp optimizer, the hyperparameters are tuned as seen in Figure 32, and the respective values as shown in Table 12 are selected, as they produce lower model loss and higher accuracy.

$$\begin{aligned} E[g^2]_t &= \rho E[g^2]_{t-1} + (1-\rho) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{14}$$

Here, $E[g^2]_t$ refers to the moving average of squared gradients, ρ is the memory decay rate, $E[g^2]_{t-1}$ is the moving average of squared gradients of the previous step, θ refers to the parameters such as weights and bias, η is the learning rate, ϵ is the parameter for numerical stability to prevent division by zero and g_t refers to the gradient of the parameters calculated during back propagation step.

Table 12: RMSProp optimizer tuned hyperparameters at MSE and RMSE loss for ANN model

Combinations	Learning rate	Learning rate decay	Epsilon	Rho
MSE and RMSE	1e-3	1e-4	1e-7	0.9

3.7.8 Adaptive momentum Optimizer

Adaptive momentum optimizer(Adam) as seen in Equation 15[9] is an another optimization method that computes the adaptive learning rates for each parameter. In addition to storing the average of past squared gradients(v_t) as in RMSProp, Adam also keeps the average of past gradients(m_t), similar to SGDMomentum. As discussed before, momentum is considered as a ball rolling down the hill, where as Adam is like a heavy ball with friction, which prefers absolutely flat minima in the error surface. From Equation 15, m_t is the estimate of the mean or first moment and v_t is the estimate of the variance or second moment. During the initial time steps the mean and variance are biased towards zero when the decay rates are small. This is due to the fact that these moments are initialized as vectors of zeros. To counteract these biases a bias-correction mechanism is applied in both mean($\hat{m}_t$) and variance($\hat{v}_t$). The corrected

moments speeds up the training in the initial epochs, and eventually matches the original moments(m_t and v_t). These corrected moments are then used to update the parameters(θ_{t+1}), similar to what was observed in RMSProp. Here, in the update step, the corrected mean is divided by the square root of the corrected variance. For the fatigue dataset, higher accuracy and lower loss is observed when the model is trained with Adam optimizer. Here, two new hyperparameters β_1 and β_2 are used as an exponential decay rate for momentum mean and variance and the same can be seen in Equation 15. With respect to the tuned hyperparameters as seen in Figure 32, the respective values as show in Table 13 are selected as they produce lower model loss and higher accuracy.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
\tag{15}
$$

Here, m_t and v_t are the estimates of first and second moments, β_1 and β_2 are the exponential decay rate for mean and variance, $\hat{m}_t$ and $\hat{v}_t$ are the corrected moments, θ refers to the parameters such as weights and bias, η is the learning rate and g_t refers to the gradient of the parameters calculated during back propagation step.

Table 13: Adam optimizer tuned hyperparameters at MSE and RMSE loss for ANN model

Combinations	Learning rate	Learning rate decay	Epsilon	Beta 1	Beta 2
MSE and RMSE	1e-3	1e-5	1e-7	0.9	0.999

3.7.9 ANN Implementation

As discussed before, the implemented MLR model has adopted the design and the structure of ANN. Thus, the work flow of the implemented functions are almost similar in both the models. In this section, the functions discussed will be an extension of what was discussed in the MLR model and the same can be observed from Figure 18.

ANN Data preprocessing: The data preprocessing is the first step of the implemented ANN model and it is similar to what was discussed in Sect.3.6.7 with slight modifications. Unlike in MLR, the training set(X_train and y_train) and the testing set(X_test and y_test) of the independent features are not filled with ones in the first column, instead, these sets are directly used for model training and prediction.

ANN model training: The implemented training procedure is an extension of what was discussed in Sect.3.6.8. In ANN model training, not only the loss metric but also the optimizer must be selected to train the model, and the same is achieved based on user input. In click, there are four options for the user to select as four optimizers are implemented(discussed in previous sections), from which one must be selected. If non of the options are selected, SGD optimizer will be considered by default, and if some other options are given as input, the program will exit with an error message: "Error: Recheck *optimizer* input; Possible inputs: SGD, SGDM, RMSP or Adam". In addition to the model loss computed using loss metrics, regularization loss is also computed and the same is added to the model loss. This in turn will be the total loss

of the training step. While writing the updated parameters for prediction using pure predictor, parameters from each layer will be considered. Note that, hyperparameters input for optimizer and regularization loss must be provided to train the ANN model.

ANN model prediction and pure predictor: The intuition behind the ANN model prediction is same as what is discussed in Sect.3.6.9, with only difference in the generated results, file names and the directories to which the computed results are stored. While executed as pure predictor, the procedure is same as specified in Sect.3.6.10 with some changes in the implementation front. The first column of the scaled dataset($scaled\_X$) used for prediction will not be inserted with ones. Similarly, dense layer function as discussed in Sect.3.7.2 is not used during the pure prediction step. Instead, a new function, updated layer is used to execute pure predictor. In this new function, the updated parameters from each layer is used as input to make predictions for new datasets. Here, new dataset refers to dataset similar to that or an extension of the training set. When pure predictor is executed from create overall results as in Sect.3.7.10, prediction results are created for eight combinations of loss and optimizer and are stored in the directory **Results_ANN_predictor**.

3.7.10 Role of creator and combination code

Create overall results: The *PPP_ANN_createOverallresults.py* is used to create results for eight combinations of loss functions and optimizers for a given input dataset. In the implemented ANN, four optimizers(SGD, SGDM, RMSP, Adam) and two loss functions(MSE, RMSE) are used. Each each optimizers has its own set of hyperparameters, and here, eight combinations can be assigned in the same time while creating the overall results. The implementation part is same as discussed in Sect.3.6.11. In addition, the code can also perform hyperparameter tuning of the neurons in the hidden layers for four different combinations as seen in *'hl1list'* from the Table 14. This simplifies the process of selecting suitable neurons for each hidden layer. The advantage of this code is it makes hyperparameter tunning of eight combinations easier and validate the ANN model for a given dataset than tuning each combination separately for different cases. It creates results of all combination, which are then later used in *PPP_ANN_combination.py* to create combination plots for analysing the model. As in MLR, ANN can also execute as a pure predictor using the trained parameters computed for eight combinations and notes the time taken for each combinations and all the combinations to execute. The options of create overall results for a given input dataset can be referred from Table 9 and Table 14 with respect to the column program used.

Combination: The *PPP_ANN_combination.py* should be executed after *PPP_ANN_createOverallresults.py* to create hyperparameter tuning plots as seen in Sect.4.2.4. In addition, the program also creates time taken results as seen in Figure 37, overall results plot as depicted in Figure 35, model results for all combination as seen in Figure 36, comparison excel and overall testing results. The purpose of the comparison excel is discussed in more detail on the combination part of the Sect.3.6.11.

3.7.11 Generalization of ANN model and other details

- The implemented ANN model is design to work for any given dataset that complies the mention set of conditions for dataset.
- As a part of generalization **click** library is used as seen in Table 14 to adjust certain user defined inputs. The click options used in the MLR model as seen in Table 9 is also applicable to ANN. These inputs can be given in the command line while executing the

program, if not given, default values are set for every input and the same can be seen in Table 14. To get more information on input options, try *filename.py - -help*.

- In addition, the hyperparameter inputs for optimizers can be given separately for each optimizer. The input values of each optimizer can be found in optimizer section in form of a table.
- Each time when a plot is created or a file is written the program automatically creates a directory if one does not exist and store the results and the same can be found in the Table 15.
- All created features in the library *PPP_ANN.py* are imported into the driver *PPP_ANN_main.py* and the input datasets must be present in the same directory as the programming files.
- This confirms the completion of the task Artificial neural networks as specified in the proposal.

Table 14: Python implementation of click in ANN with default values and the required input data types for all the three programs except the library. This is additional to what is implemented in MLR as seen in Table 9

click.option	nargs	dtype	default	help	Program used
- -optimizer	1	str	Adam	Select optimizer: [SGD,SGDM,RMSP or Adam]	All the three
- -layers	2	int	[10, 10]	Enter hidden layer(N1, N2) input for [IP-N1-N2-OP](>0) based on HP tuning	All the three
- -reg	6	float	[2e-6, 2e-6, 3e-6, 3e-6, 0, 0]	Enter regularization loss(L2) for the layers	driver and overall
- -opt	2 to 5	float	refer table 7, 11 to 13	Enter optimizer input @MSE	All the three
- -opt_rmse	2 to 5	float	refer table 7, 11 to 13	Enter optimizer input @RMSE	Overall and combination
- -optimizerlist	4	str	[SGD, SGDM, RMSP, Adam]	Optimizer input as a list	Overall and combination
- -writehl	1	int	1	Select 0 or 1 to write hidden layer results	driver and overall
- -hl1list	4	int	[70, 50, 20, 10]	Enter hidden layer1 list(N1) for HP tuning	Overall and combination
- -hl2list	4	int	[70, 50, 20, 10]	Enter hidden layer2 list(N2) for HP tuning	Overall and combination
- -selectdata	6	int	[500, 50, 500, 50, 500, 20]	Selecting value range to generate readable plots for Accuracy_loss as in Figure 32, learning rate as in Figure 34 and hidden layer as in Figure 33	combination

3.8 k-Nearest Neighbors for Regression

k-Nearest Neighbors(kNN) is the third and the last regression model that is implemented to predict the fatigue strength of steel. Like in the previous cases, kNN regression is also an supervised machine learning model to make predictions. It can be considered as an general approximator that is entirely based on the patterns present in the data, without any specific

Table 15: Files and directories created during ANN implementation and the highlighted ones are created during *PPP_ANN_combination.py*

Directory	Files and plots	Remarks
Plots_ANN_Main	plot_testingerror_Loss_Opt.png plot_testingresults_Loss_Opt.png plot_trainingresults_Loss_Opt.png	Plots of Training and testing results and testing error
Plots_ANN_combination	plot_LossatHL.png plot_Overall_Accuracy.png plot_Overall_Loss.png plot_Overall_LearningRate.png plot_Overall_TestingResult_Loss.png plotOverall_results_Loss.png	Hyperparameter tuning of hidden layer as seen in Figure 33, tuning of optimizers w.r.t accuracy and loss as seen in Figure 32; Learning rate intuition as seen in Figure 34; Overall prediction results and Overall results plot as depicted in Figure 35
Results_Training	trainingresults_Loss_Opt.txt	Tabulated training results at MSE/RMSE and SGD/SGDM/RMSP/Adam
Results_Testing	testingresults_Loss_Opt.txt	Testing results at MSE/RMSE and SGD/SGDM/RMSP/Adam
Results_Parameters	parametersANN_Loss_Opt.npy/txt	Trained parameters
Results_ANN_predictor	testingresults_Loss_Opt.txt resultcomparison_Loss_Opt.txt	Testing results and results comparison table for all loss and optimizer combination is created during pure predictor
Results_TargetVSpred	resultcomparison_Loss_Opt.txt resultcomparisonANN.xlsx resultModelperformance_ANN.txt timetaken_overall.txt	Result comparison table for all loss and optimizer combination, overall results comparison Excel, model performance analysis data as seen in Figure 36 and time taken in create overall results
Results_Timetaken	resulttimetakenANN.txt	Time taken results of the model for each dataset as seen in Figure 37
Results_forPlotting	lossatHiddLay10101_Loss_Opt.txt plotdata_Loss_Opt.txt	Results created in all combinations for analysis

Table 16: Functionalities implemented in the kNN model

Functionality Implemented	Remarks	Functionality Implemented	Remarks
Import dataset	File format: .csv	Determine k neighbors	Creates loss plot to select neighbors(k)
Feature scaling	Standardizing features	Time taken	Time taken results of the model for each k input
Test-train split	80% training set, 20% testing set	Loss functions	MSE and RMSE
Predict neighbors	Makes Prediction; Writes and plots prediction results	Coefficient of determination	$R^2 = 1$, for best case fit

statistical model that must be estimated. The kNN method does not depend on any model, instead it uses the input dataset to predict new observations. While predicting a new observation, the algorithm find most similar observations from the main dataset, and uses the predictions of these observations to predict the output. The kNN implementation is programmed into two files with one acting as a library implemented with the required functions and other is the driver which performs data preprocessing, choose nearest neighbors(k) and make predictions. The features implemented in the kNN model can be found in Table 16, and the details of which will be discussed in the upcoming sections.

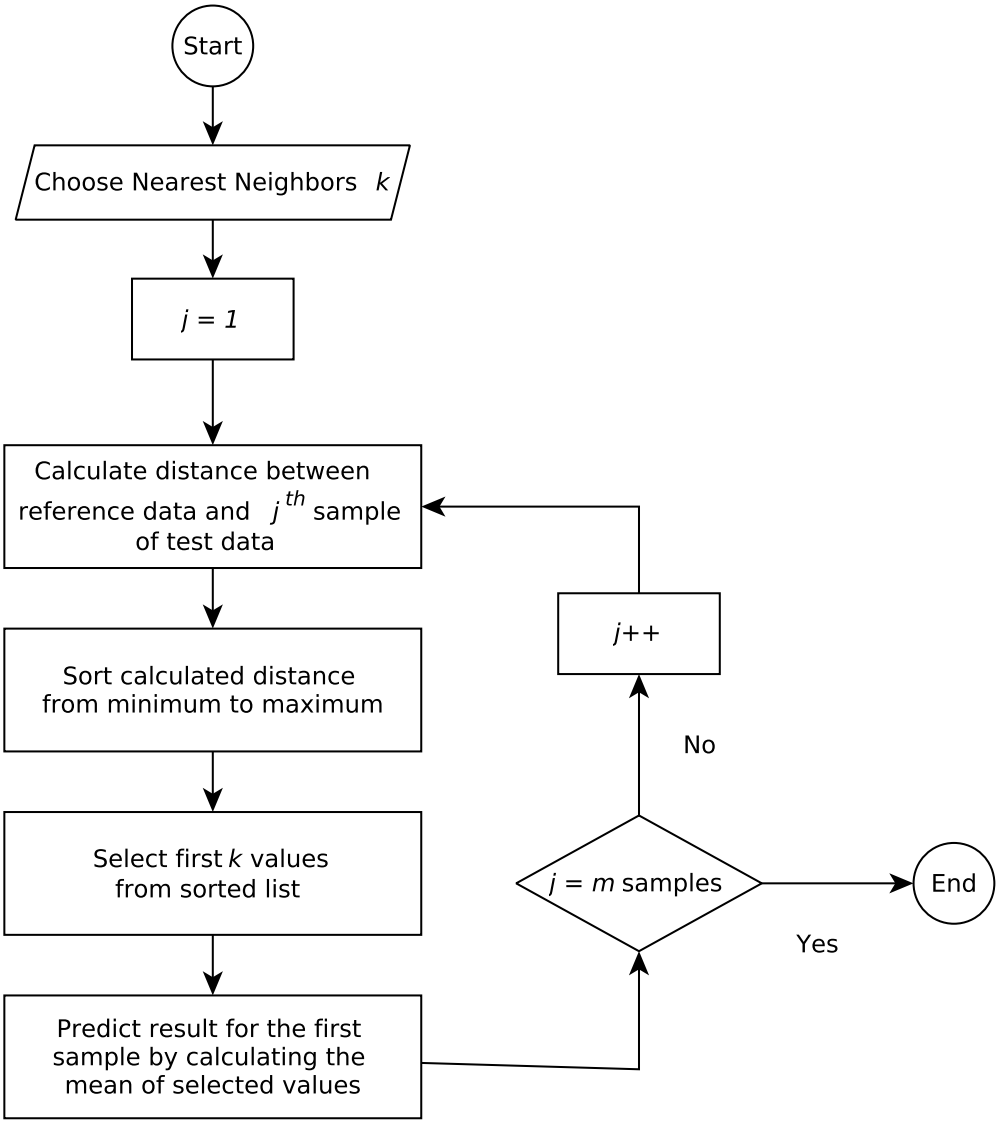


Figure 21: Flowchart explaining the working of the k-Nearest Neighbors algorithm.

3.8.1 Working of kNN algorithm

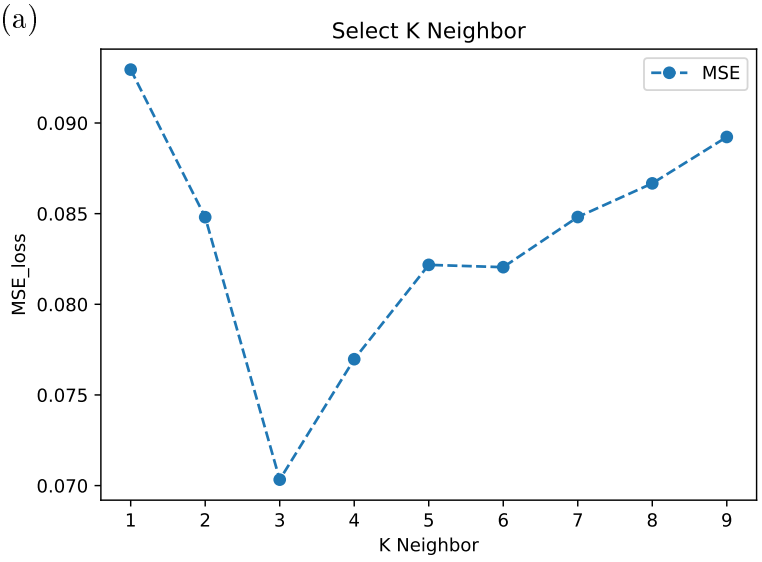
In k-Nearest Neighbors(kNN) the prediction of each observation is based on the selected number of nearest neighbors(k). The dataset used is split into training and validation set, in which the former will be the main data with n observations and later be the test data with m observations($m < n$). Both the main and the test datasets have independent and dependent features. For each j^{th} sample of the m observations, similar set of data are identified from the main dataset. The fatigue dataset consist of 437 observations, in which 349 is present in the main data and 88 in the test set. To find out the similar observations for the j^{th} sample, euclidean distance between the n observations of the main set and the j^{th} sample is computed. The computed distance are then sorted from minimum to maximum, and the number of observations that are closer to the j^{th} sample is selected based on the chosen nearest neighbors. In the implementation, the distance between 349 observations of main set and one of the 88 observations of test set is computed. The distance result of 349 observation are then sorted in ascending. For number of nearest neighbors three, the first three observations from the sorted list will be selected and their dependent features are noted. These selected observations are then used to predict the fatigue strength of the first test sample by computing the mean of the

noted dependent features(fatigue strength). This process is then repeated for all the samples of test set and the prediction results for the same will be noted. The algorithmic working of the kNN algorithm is depicted in Figure 21.

3.8.2 Choose nearest neighbors

The prediction accuracy of the kNN model depends on the number of nearest neighbors(k). In the previous section, the test set is predicted at number of nearest neighbors three, but to select the optimal neighbors, the model is predicted using a predefined set of neighbor inputs. The predicted results are then used to compute the model loss for each k neighbor. The k neighbor that results in the lowest model loss is considered as the optimal neighbor and the same will be printed as the program output. The predefined set are the nearest neighbor values ranging from one to end limit, where, end limit is an user input and by default assigned as nine. The user input of the end limit also referred to as $kfind$ in click and should be greater than zero, if not the program exits with an error: "Error: $kfind$ input must be > 0". The model loss is computed using MSE/RMSE loss function, the loss is also selected based on user input and by default assigned as MSE. The optimal number of neighbors with lowest model loss can also be selected by constructing a loss plot using the predefined set of neighbors against its respective loss value and the same is depicted in Figure 22(a). This optimal number of neighbors are then used for kNN model prediction, more details on prediction results and analysis will be discussed in the result analysis section(Sect.4.2.6).

(a)



(b) Time taken in seconds for prediction in KNN at different nearest neighbors(K) inputs:

K	Time
1	0.002419
2	0.0025221
3	0.0025245
4	0.0025727
5	0.0025516
6	0.0034488
7	0.0032804
8	0.0030619
9	0.0024859

Time taken by the model to execute: 0.538298099

Figure 22: (a)Plot to select optimal number of neighbors for prediction in k-Nearest Neighbors;(b) time taken to predict the validation set for each k input.

3.8.3 kNN model prediction

The data preprocessing step in the kNN model is similar to what was discussed in the MLR model, excluding the part where y-intercept of the independent features being filled with ones. Unlike MLR or ANN the model prediction by kNN is quit simple and faster with little compromise in model accuracy. In the data preprocessing step the fatigue dataset is split into training and validation data with independent and dependent features. The split dataset along side with the predefined set of neighbors(k) are used to determine the optimal number of neighbors. In case of the fatigue dataset the optimal number of nearest neighbors for kNN model is three and

the same is observed in Figure 22(a) and in program output. The determined neighbors(three) are then used to make kNN model prediction of the validation set. To compute the accuracy of the predicted results, coefficient of determination is used as discussed in the Sect.3.6.5. To compute the model loss, MSE or RMSE loss is used as discussed in the Sect.3.6.3 and 3.6.4. The rescaled prediction results along with accuracy and loss values are written into the file *resultcomparision.txt*.

Table 17: Files and directories created during kNN implementation

Directory	Files and plots	Remark
Plots_KNN_main	plot_Kneighbor.png plot_PredictionResults.png plot_PredictionError.png	Plot to determine nearest neighbors(k) as seen in Figure 22(a); Plots of prediction results as seen in Figure 38 and prediction error
Results_TargetVSpred	resultcomparision.txt	Resut comparision table of target and predicted results
Results_Timetaken	resulttimetakenKNN.txt	Time taken results of the model for each k input as seen in Figure 22(b)

Table 18: Files and directories created during Overall-results

Directory	Files and plots	Remark
PPP_Overallresults	plotOverall_results.png resultcomparisionOverall.xlsx resultModelperformance.txt	Plot to analyse best performing models as seen in Figure 40, Writing the prediction results of best performing models into a .xlsx file, And the model performance result for each model as seen in Table 28

3.8.4 Generalization of kNN model and other details

- The implemented kNN model is design to work for any given dataset that complies the mentioned set of conditions for dataset.
- As a part of generalization **click** library is used as seen in Table 19 to adjust certain user defined inputs. These inputs can be given in the command line while executing the program, if not given, default values are set for every input and same can be seen in Table 19. To get more information on input options, try *filename.py - -help*.
- Each time when a plot is created or a file is written the program automatically creates a directory if one does not exist and store the results and the same can be found in the Table 17
- All created features in the library *PPP_KNN.py* are imported into the driver *PPP_KNN_main.py* and the input datasets must be present in the same directory as the programming files.
- This confirms the completion of the task k-Nearest Neighbors as specified in the proposal.

Table 19: Python implementation of click in kNN with default values and the required input data types

click.option	nargs	dtype	default	help
- -data	1	str	fatigue_dataset.csv	Enter input dataset .csv: last column must be the target feature
- -loss	1	str	MSE	Select loss: [MSE or RMSE]
- -kfind	1	int	9	Enter end limit for no of neighbors in Figure 22(a)
- -k	1	int	3	Selected no of neighbors from Figure 22(a)

3.9 Overall results

Three different predictive models, namely, MLR, ANN, kNN are used to predict the fatigue strength of steel. After validating each model i.e., after executing all the program files present in each model, required results are created. Now, a separate program file, *PPP_ Overallresults.py* is used to create overall results with the results generated from all the models. Which means, this program should be executed only after executing all the model. The overall-results program helps us to analyse the best performing combination from each model by creating result analysis table, plot and an excel file with prediction results of each model. The best performing combination from each model is selected based on user input, and the same can observed in Table 20. Certain conditions with respect to user inputs are listed below. The details on the created results will be discussed in Sect.4.2.7.

- As in Table 20, for *models* input, the user must specify the models in the given order, with some attention on ANN where the user should additionally specify optimizer detail as ANN_optimizer. Here, the optimizer details shall be selected from the options specified in the *optimizer*. By default ANN_Adam is selected and the same is observed in the first row of the Table 20.
- Additionally, the user should provide details on which loss metric to use for each model and which optimizer should be selected for ANN. The optimizer selected should be same as the optimizer selected in the *models*.
- Other details regarding the number of samples in the validation/test set, the name of the target feature that we are predicting and the lower and upper bound of the computed accuracy and loss values(from three cases) should be specified. These values are then used as ylimits in overall results plot depicted in Figure 40.

3.10 Proposed VS Complied

This section is to verify the completion of the proposed task. In data assessment step, PCA and correlation matrix is additional to what was proposed. In K-mean clustering, elbow method is used to select number of clusters and K-means algorithm is applied to make clusters. In the task of MLR model, loss function, optimizer function and accuracy function is completed. In case of ANN model, all tasked specified in Table 21 is completed. In addition, the results analysis of all model is also completed. The details of the tasks and reference to the completed sections are provided in Table 21.

Table 20: Python implementation of click in Overall-results with default values and the required input data types

click.option	nargs	dtype	default	help
- -models	3	str	[ANN_Adam, MLR, KNN]	Enter model input(Example: ANN_Optimizer)
- -loss	1	str	MSE	Select loss: [MSE or RMSE] for ANN
- -lossmlr	1	str	MSE	Select loss: [MSE or RMSE] for MLR
- -lossknn	1	str	MSE	Select loss: [MSE or RMSE] for KNN
- -optimizer	1	str	Adam	Select optimizer: [SGD, SGDM, RMSP or Adam] for ANN
- -len_test	1	int	88	Enter the no of samples in test case (i.e. 20 percent of the total sample)
- -targetf	1	str	Fatigue Strength	Enter the name of the target feature
- -limacc	2	float	[0.93, 0.99]	Enter lower and upper bound for accuracy values
- -limls	2	float	[0.02, 0.08]	Enter lower and upper bound for loss values

Table 21: Details of the tasks proposed and complied. The highlighted ones are additional to what was proposed

Proposed	Complied(Yes/No)	Reference
Data assessment PCA K-means Clustering Correlation Matrix	yes	Refer Section.3.2
Multi Linear Regression Loss function Optimizer function Coefficient of determination	yes	Refer Section.3.6
Artificial Neural Networks Feed forward function Activation function Regularization loss Loss function Coefficient of determination Back propagation function Optimizer function Hyperparameter tuning	yes	Refer Section.3.7
k-Nearest Neighbors Choose nearest neighbors	yes	Refer Section.3.8
Performance Analysis	yes	Refer Section.4.1

3.11 Software and Libraries used

The software and libraries used in this project are listed in Table 22. It is recommended to adhere with the specified versions of the libraries in order to avoid any possible mismatch in the solutions. The programming language used in this project is python. The tasks can also be programmed in other languages like, R, Java and Julia. Python is preferred due to its simple syntax and better code readability. In addition, python also has versatile libraries for scientific computation and visualization. This is one of the reasons to prefer python over other languages.

The machine used for computation in this project consists of four cores and eight threads and it is powered by 11th Generation Intel i7 Processors with 16 GB RAM.

Table 22: Software and libraries used in this project. These are standard and external libraries that can be installed from command line with pip.

Library/Software	Version	Remarks	Library/Software	Version	Remarks
Language: Python	3.9.5	Editor used: Visual Studio Code, version 1.64.2 [link]	sys	3.9.5	To exit the program and to get the absolute path of the executable
numpy	1.21.3	For mathematical operations	os	3.9.5	To make directories
scipy.linalg	1.7.1	To calculate eigen values and vectors in PCA	subprocess	3.9.5	Returns the executed code of the program
pandas	1.3.3	For data analysis and manipulation	click	8.0.1	To create command line interface
matplotlib.pyplot	3.4.3	To make plots	timeit	3.9.5	To measure execution time
seaborn	0.11.2	To make heatmaps in CMatrix	pytest	6.2.5	To test implemented functions
XlsxWriter	3.0.2	To write into a worksheet	tabulate	0.8.9	To make tables

4 Results of tests and discussion

4.1 Testing details

The test cases and program files of all tested functionalities are present in the directory *test_ML*. Each tested functionalities are stored in separate directories with a *README.txt* file specified with details on how the functionality is tested. With respect to the test case, the directories also contain datasets and *parameters.txt* files with details on input set and expected output. The test functionalities are asserted using pytest framework and the same can be seen in Figure 23. To test all functionalities in a single go, execute pytest -l (prints less details) / pytest -v (prints more details) from the directory *test_ML*. To test individual functions, execute the same from the functions directory. In some cases the test files can also be executed normally as 'python testfile.py'.

```
(ml) PS D:\1.CMS\1.Class Notes\Sem_3_2021\1.PPP\Program\test_ML> pytest -l
========================================= test session starts =========================================
platform win32 -- Python 3.9.5, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: D:\1.CMS\1.Class Notes\Sem_3_2021\1.PPP\Program\test_ML
collected 56 items

test_ANN\test_ANN_Back_prop\test_ANN_Back_prop.py .....                                          [  8%]
test_ANN\test_ANN_Forward_prop\test_ANN_Forward_prop.py .....                                    [ 17%]
test_ANN\test_ANN_Functions\test_ANN_Functions.py ..........                                     [ 39%]
test_ANN\test_ANN_Overfitting\test_ANN_Overfitting.py .                                          [ 41%]
test_ANN\test_CompleteANN_Adam_optimizer\test_CompleteANN_Adam_optimizer.py .....                [ 50%]
test_ANN\test_CompleteANN_RMSProp_optimizer\test_CompleteANN_RMSProp_optimizer.py .....          [ 58%]
test_ANN\test_CompleteANN_SGDMomentum_optimizer\test_CompleteANN_SGDMomentum_optimizer.py .....  [ 67%]
test_ANN\test_CompleteANN_SGD_optimizer\test_CompleteANN_SGD_optimizer.py .....                  [ 76%]
test_KNN\test_KNN_Prediction.py .....                                                            [ 85%]
test_Kmeans\test_KMeans_Clustering.py .                                                          [ 87%]
test_MLR\test_CompleteMLR\test_CompleteMLR.py .....                                              [ 96%]
test_MLR\test_MLR_Overfitting\test_MLR_Overfitting.py .                                          [ 98%]
test_PCA\test_PCA.py .                                                                           [100%]

========================================= 56 passed in 1.37s ==========================================
```

Figure 23: Testing results of implemented functionalities tested using pytest framework.

4.1.1 Test PCA

There are two methods to validate the Principal Component Analysis[3]. First, verify whether the eigen values obtained from the covariance matrix is equal to the total variance of the input data i.e., equal to the sum of the diagonal elements of the covariance matrix. Second, recall that PCA are the eigen vectors of the covariance matrix and by definition the length of the computed eigen vector should be one[3]. Both of this tests are satisfied by the implemented PCA. In addition, the PCA is also validated with a sample dataset from E-Learning Project SOGA [3]. The results obtained for this sample dataset matches the original results present in E-Learning Project and same is depicted in Figure 24. These results include, Biplot, Scree plot and PCA results. More details on the test and other obtained results can be found in the subdirectory *test_ PCA*.

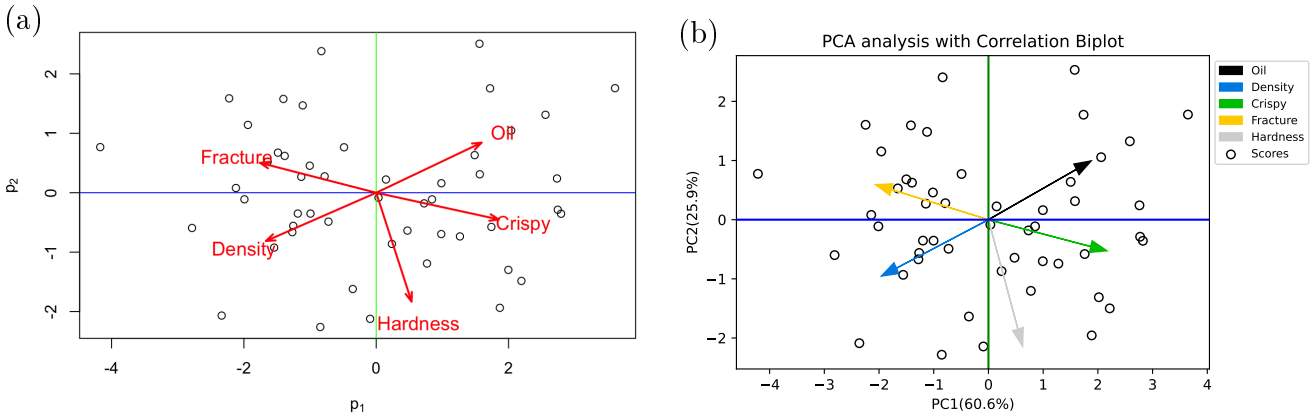


Figure 24: PCA results of Biplot for the sample dataset,(a)expected result referred from E-Learning Project SOGA [3]; (b) results obtained from the implemented PCA.

4.1.2 Test K-means

Unlike in PCA, there is no legitimate procedure to validate the K-means clustering method. Therefore, the implemented K-means is validated using a sample dataset from digital publishing domain medium[13]. The results obtained by the implemented K-means for the sample dataset matches the original results present in the article and same is depicted in Figure 25. Here, the number of clusters created is five, and the same is observed on the elbow plot for the sample dataset. Similarly, the implemented K-means is also tested for three other sample datasets and the results obtained matches with the expected results. This verifies the correctness of the implemented K-means. For more details regarding the sample datasets and the obtained results can be found in the subdirectory *test_ Kmeans*.

4.1.3 Test ANN

Forward Propagation: The forward propagation step of the neural network is validated using know inputs and pre-calculated output values. The input set consist of five single samples with each sample having five neurons. The inputs for the weights and biases of each layer and the number neurons in the hidden layer are fixed. More details on the test can be found in the subdirectory *test_ ANN_ Forward_ prop*.

Backward Propagation: The backward propagation(BP) step of the neural network is tested using the **gradient checking** method[14]. Here, the numerical gradient as seen in Equation 16 verifies, whether the analytical gradient computed during the BP step of the NN matches with the numerically computed gradient. This numerical gradient is estimated using the central

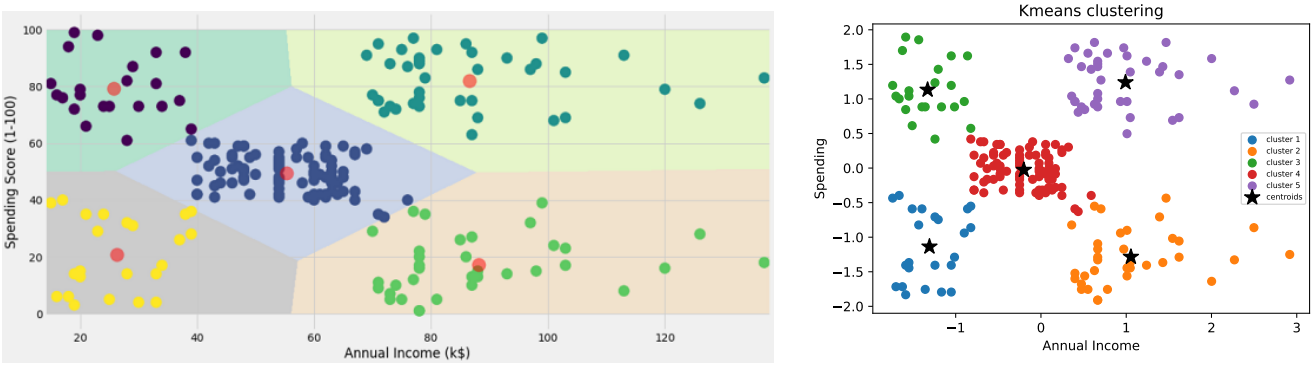


Figure 25: K-means clustering results for the sample dataset; (left)expected result referred from digital publishing domain medium[13]; (right)results obtained from the implemented K-means.

Table 23: Details on test performed on the implemented ANN model and the test results.

Functionality tested	Number of test cases	Passed(Yes/No)
Neural network assembly		
Forward propagation	5	yes
Backward propagation	5	yes
Overfitting using single dataset	1	yes
Training with single sample	20	yes
Unit-tests		
Activation functions	4	yes
Loss and accuracy functions	8	yes

difference method by introducing small perturbations. Each function in the BP step is verified by computing its forward propagation step with inputs perturbed positively and negatively as seen in Equation 16. In ANN we back propagate from loss_function → activation_function → dense_layer, thus, multiplied by the derivatives of the previous functions due to chain rule. More details on the test can be found in the subdirectory *test_ANN_Back_prop*.

$$\frac{\partial J}{\partial \theta} = \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon} \tag{16}$$

Here, J refers to the forward propagation function, θ is the input to the function and ϵ is the small perturbation of 0.001.

Training with single sample: Validating one complete training step with single sample data using gradient checking method for different optimizers. In a single training step the optimizer updates the parameters like weights and biases with the derivatives computed during the back propagation step. Similar to the back propagation test, here to the derivatives are computed using the numerical gradients method. During the optimization step, the parameters update computed with the analytical method is verified with the parameters update computed using the numerical gradients. This test is performed separately for each optimizer. More details on the test can be found in the subdirectory *test_CompleteANN_X_optimizer*, here X denotes the type of optimizer.

Overfitting using single dataset: The model is trained with the input dataset, after training, the same dataset is used again to validate the model. The resultant prediction accuracy should 100% match the training accuracy, i.e., the implemented ANN model should over-fit the dataset

with 100% training accuracy. More details on the test can be found in the subdirectory *test_ANN_Overfitting*.

Unit testing: Each functions used in the ANN model was tested using known inputs and pre-calculated outputs. These functions include, activation function, loss metrics, regularization loss and coefficient of determination. More details on the input set and parameter set can be found in the subdirectory *test_ANN_Functions*.

4.1.4 Test MLR

Table 24: Details on test performed on the implemented MLR model and the test results.

Functionality tested	Number of test cases	Passed(Yes/No)
Multi linear regression assembly		
Forward propagation	5	yes
Backward propagation	5	yes
Overfitting using single dataset	1	yes
Training with single sample	5	yes

As the multi linear regression model implemented in this project is adapted from the program design of the neural networks, all the functionalities implemented in the MLR fallows the testing procedure of the neural network(Sect.4.1.3), and the same can be observed from Table 24. More details on input dataset and parameter set can be found in the subdirectories *test_MLR*.

4.1.5 Test KNN

The kNN testing is based on the working of the kNN algorithm discussed in Sect.3.8.1. To test the kNN model, a parameter set is created similar to other models with independent features, dependent features, expected results and number of nearest neighbors. The input features are start and end value of numpy.arange, from which a dataset is created to test kNN prediction. The test is done for five different cases, with each case containing atleast six samples. As mentioned in Sect.3.8.1, the dataset is split into training and validation set, and the distance between each test case and the samples in the training set is computed. The computed distance are then sorted(min to max), and the observations similar to the test case are selected based on nearest neighbors input from the parameter set. The mean of the selected observations are computed and the result of the same will be the prediction result of the test case. In kNN, the output results are pre-calculated for known inputs and are then used to verify the results obtained from the kNN model. More details on the test can be found in the subdirectory *test_KNN*.

4.2 Result analysis

4.2.1 PCA result interpretation

In the derive new features section(Sect.3.3.4) we have derived the score vector i.e., derived by computing score value of each observations. These scores along with the loading vectors are plotted together as Biplot and the same is depicted in Figure 26. The score value for an observation is the point where that observation projects onto the direction vector for say the first component(*Loading_vector*1)[3]. An important point with PCA is that the projection

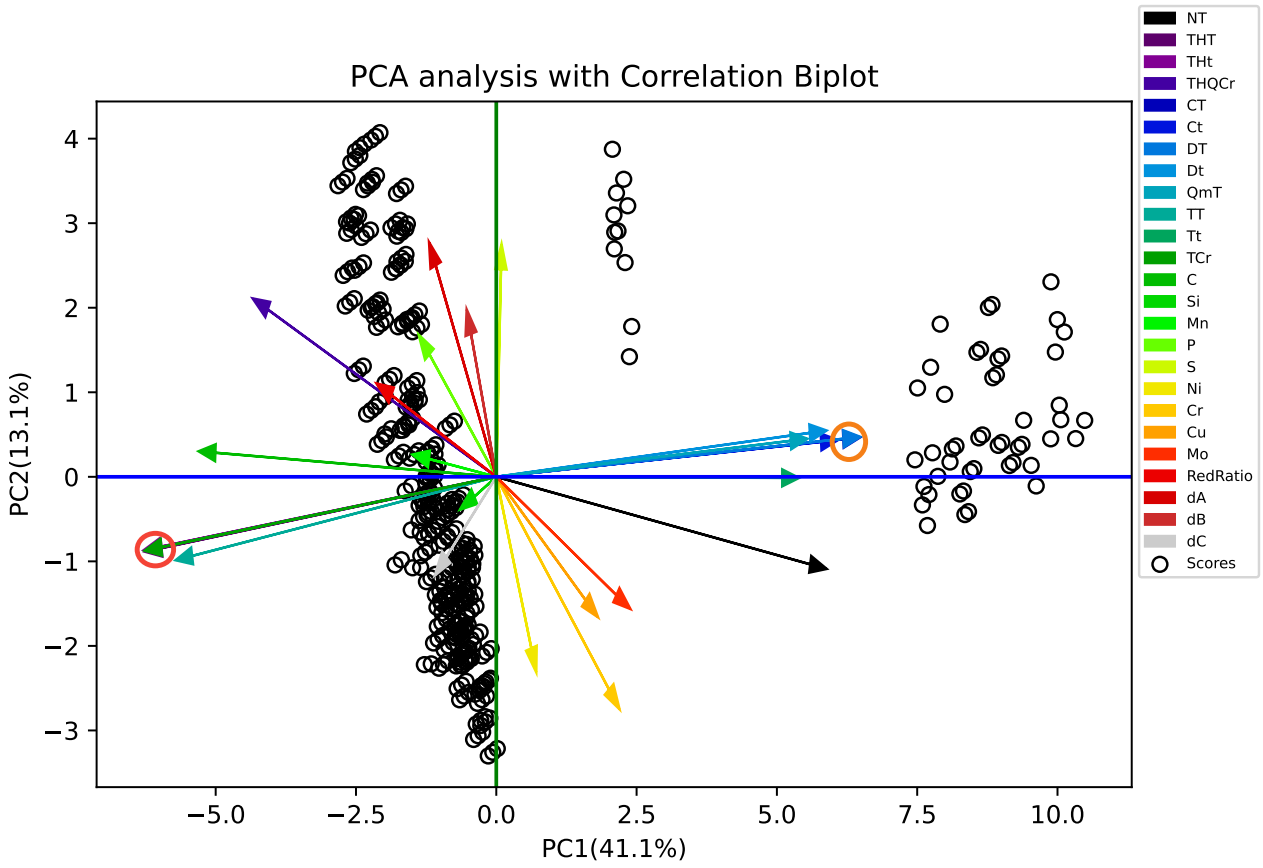


Figure 26: Interpreting PCA results of dimensionality reduction with biplot for all the 25 features.

matrix(loading matrix) is orthonormal, any relationships that were present in scaled features are still present in the score vector. Thus, score scatters present in the biplot allow us to rapidly locate similar observations, clusters, outliers and time-based patterns. The first two score vectors(PC1 and PC2) explains the greatest variation in the data, hence it is considered for the scatter plot of the scores in the biplot and the same is depicted in Figure 26[3]. Due to the larger number of observations(437) the scores in the biplot are not named, thus, making it difficult to locate the i^{th} score of the n observations. But, the extracted dataset that was written into a csv file will be of use as it has the coordinates of the scores present in the biplot. The loading vectors in the biplot are the direction vectors that define the model and these vectors begin at the origin and extend to the coordinates. If it is difficult to interpret the loading vectors(features) based on their color scheme, the *LoadingResults.txt* file can be used. This file has the vector coordinates of each features and thus, can be used together with the biplot for accurate interpretation. The coordinates of 13 features out of 25 is shown in Figure 27 and will be used for result interpretation in this section.

Analysing scores and loading vectors in Biplot[3]:

- **Claim:** The features which have little contribution to a loading vector have almost zero weight in that loading and the loading vectors are linearly uncorrelated to each other. **Observation:** In this project there are two loading vector as in Figure 27 with a variance of 41% and 13%(marked with green) respectively. This means the features that has contributed to loading vector 1(marked with red) has contributed with higher weightage than the features contributed to loading vector 2 and the same shall be observed

Table 25: Details on features that have contribution and no contribution on loading vector 1 and 2

Feature contribution	Feature detail	No of features
Features contributing to loading vector 1	NT, THT, THt, THQCr, CT, Ct, DT, Dt, QmT, TT, Tt, TCr, C, Mo	14
Features with little or no contributing to loading vector 1	Si, Mn, P, S, Ni, Cr, Cu, RedRatio, dA, dB, dC	11
Features contributing to loading vector 2	NT, THT, THt, THQCr, TT, TCr, P, S, Ni, Cr, Cu, Mo, RedRatio, dA, dB, dC	16
Features with little or no contributing to loading vector 2	CT, Ct, DT, Dt, QmT, Tt, C, Si, Mn	9

from the results provided in Figure 27. To make the analysis easier a table is created as seen in Table 25 based on *LoadingResults.txt*. This table contains the details of features that are contributing to the loading vector 1 and 2, and features that are not. From the observation in Table 25 fallowing conclusion can be made. Most of the features that have little or no contribution in loading vector 1 have good contribution in loading vector 2 and vice versa. This confirm the claim of loading vectors being linearly uncorrelated to each other and the same is also confirmed by the correlation plot as seen in Figure 12(b). The features that have little contribution to the loading vector, example, 'Sulphur(S)' in loading vector 1 and 'Tempering Time(Tt)' in loading vector 2 have almost zero weights. Some of the features like 'Silicon(Si)' and 'Manganese(Mn)' as seen in Table 25 do not contribute to both the loading vectors and are likely to contribute to other loading vectors that exists.

- **Claim:** Features which have roughly equal influence on defining a direction are correlated with each other and will have roughly equal numeric weights. **Observation:** Form Figure 27 we can identify the features that have roughly equal numeric weights in a direction. Example, 'Through Hardening Temperature(THT)', 'Through Hardening Time(THt)' and 'Cooling Rate for Tempering(TCr)' in loading vector 1(marked with red) have almost same weights. Based on the correlation plot present in Figure 9 we can confirm that these feature with similar weights have 100% correlation with one another.

- **Claim:** Strongly correlated features, will have approximately the same weight value when they are positively correlated. In a biplot they will appear near each other, while negatively correlated features will appear diagonally opposite to each other. **Observation:** Here, there are two combinations in loading vector 1. First, the THT, THt and TCr, second, 'Carburization Temperature(CT)' and 'Diffusion Temperature(DT)'. The features in the combinations are 100% positively correlated with each other and also has approximately the same weights as observed in the Figure 27. As seen in Figure 26, the feature vector of CT and DT present in top right region(circled in orange) overlaps with each other. Similarly, THT,THt and TCr present in bottom left region(circled in red) overlaps with each other and confirms the claim of strongly correlated features appear near each other. Where as THT and CT(both circled) are negatively correlated with the correlation of -0.89 as seen in Figure 9 and are opposite to each other.

- The orientation (direction) of the vector with respect to the principal component space, in particular, its angle with the principal component axes, the more parallel the vector is to the axis the more it contributes only to that PC. The angles between vectors of different features show their correlation in this space. Here, small angles represent high

```
Loading_vector: Eigen vectors of the covariance matrix; Max varience i.e. 1 & 2 of these vectors
Loading_value: All 25 Eigen values of the covariance matrix are listed below
Variance: It is the variance of all 25 Eigen values in percentage
Vectors of each features in the Biplot is plotted by
(x/y) -> Loading_vector[1/2] * Loading_value[0/1]/factor; factor -> 0.5

The details below are used for Biplot analysis:
```

Features	Loading_vector1	Loading_vector2	Loading_value	Variance(%)
NT	0.271553	-0.158254	10.3	41.1057
THT	-0.289583	-0.128979	3.27015	13.0507
THt	-0.290586	-0.12643	1.59835	6.37878
THQCr	-0.197681	0.302646	1.44986	5.78617
CT	0.300218	0.0680024	1.40441	5.6048
Ct	0.280965	0.0631412	1.20744	4.81869
DT	0.300385	0.0689621	1.0492	4.1872
Dt	0.271226	0.0795104	0.840752	3.35531
QmT	0.254745	0.0651141	0.702012	2.80162
TT	-0.263117	-0.14308	0.664947	2.6537
Tt	0.247367	-0.00189739	0.537926	2.14678
TCr	-0.289763	-0.127286	0.478595	1.91
C	-0.243011	0.0435496	0.360314	1.43796

Figure 27: Details of loading vector, loading value and variance for the first thirteen features to analyse Figure 26.

positive correlation(THT, THt, TCr), right angles represent lack of correlation(S, Tt with a correlation of -0.024), opposite angles represent high negative correlation(THT, CT) and the same is observed from Figure 26 and supported by Figure 9[3]. This shows that the relationship present in original data is also present in the score vector i.e., the principal components with maximum variance and confirms the correctness of the implementation.

4.2.2 K-means result interpretation

In the make cluster section(Sect.3.4.3) two clusters plot where created, one with respect to the results of the elbow plot, i.e., at number of cluster three and other for visualization at number of cluster six, the same is depicted in Figure 7. To verify whether the details of the original dataset is captured on the reduced dataset and to check for any visible patterns, clustering plots are created. The data present in the original dataset consists of three grades of steels i.e., 371 carbon and low alloy steel, 48 carburizing steel and 18 spring steel. Here, the low alloy steel and the spring steel have almost similar compositions and process parameters and thus expected to cluster together. To interpret the obtained clustering results, the result summary of the created clusters are used. To verify the result summary parallel coordinates plots[15] are utilized and both can be observed in Figure 28 and Figure 29. The Parallel coordinate plots are used to analyse multi-features numerical data. It allows to compare the features of several individual observations across multiple numerical variables(features). Each feature represented in the plot has its own axis, and axes of all the features are equally spaced and are parallel to each other. The axis of each feature can have different scale and unit of measurement. To verify the influence of each feature on the created clusters, standardised data($scaled\_X$) combined together with clustering details is used as an input dataset. That is, clustering details of each

observation from cluster results(refer cluster column) as seen in Figure 8 is combined with the standardized dataset to create parallel coordinates plot. In a parallel coordinate plot, if the feature has a weightage closer to zero, then the feature is considered to be one among the features that do not contribute to a cluster. Similarly, if the feature has positive or negative weightage, then the feature is considered to be contributing to a cluster.

```
K-means Cluster Analysis:

Number of clusters: 3
Dataset: Principal Components 1 and 2

Observation: Total number of observation for each cluster
Avg_Dist: Average distance from centroid
Max_Dist: Maximum distance from centroid

+------------+---------------+------------+------------+
| Clusters   |   Observation |   Avg_Dist |   Max_Dist |
+============+===============+============+============+
| cluster1   |           134 |   1.32737  |    4.1292  |
+------------+---------------+------------+------------+
| cluster2   |           255 |   0.819509 |    2.14367 |
+------------+---------------+------------+------------+
| cluster3   |            48 |   1.02655  |    2.00384 |
+------------+---------------+------------+------------+

Time taken by KMeans to execute(seconds): 2.92839
```

Figure 28: Result summary of K-means clustering at number of clusters three.

Result analysis of K-means clustering:

- From the result summary of both the clustering plots as seen in Figure 28 and Figure 30, the number of observations at cluster 3 is 48 and it matches with the number of carburizing steels(48) present in the original dataset. To verify whether the 48 observation present in cluster 3 is indeed carburizing steel, parallel coordinate plot is used as seen in Figure 29 and Figure 30.

- Carburizing is a heat treatment process that produces a surface that is resistant to wear and maintains toughness and strength of the core. It increases strength and wear resistance by diffusing carbon into the surface of the steel creating a case while retaining a substantially lesser hardness in the core. Carburizing is a time/temperature process where carbon potential of the gas can be lowered to permit diffusion. After carburizing, the work is slow cooled for later quench hardening and in some cases the product is tempered to lower the hardness with respect to the mechanical property requirement.

- From Figure 29 and Figure 30 one can see that the number of features contributing to cluster 3 are the same irrespective of the number of clusters. The verification of the presence of carburizing steel in cluster 3 is evaluated based on process parameters. Certain process parameter features like carbrization temperature/time(CT,Ct), diffusion temperature/time(DT,Dt) and quenching media temperature(QmT) for carburization have their weightage only in cluster 3 and not in any other clusters. This clearly shows that the 48 observations present in cluster 3 is indeed the carburizing steel observed as in original dataset. This verifies that the cluster 3 created using reduced dataset contains the carburizing sample details of the original dataset.

- The remaining clusters present in Figure 28 has the observations of carbon and low alloy steel(371) and spring steel(18) and no observation of carburizing steel. Unlike carburizing steel, there is no explicit clustering for the specified steels. Some of the entries of these

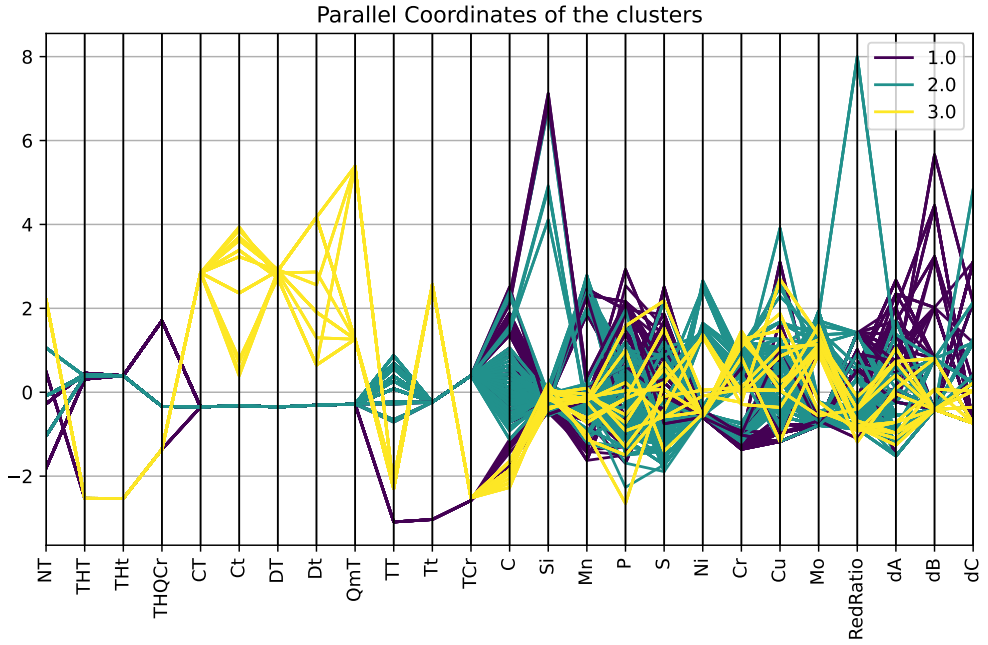


Figure 29: Interpreting K-means clustering results with parallel coordinates plot at number of clusters three.

K-means Cluster Analysis:

Number of clusters: 6
Dataset: Principal Components 1 and 2

Observation: Total number of observation for each cluster
Avg_Dist: Average distance from centroid
Max_Dist: Maximum distance from centroid

Clusters	Observation	Avg_Dist	Max_Dist
cluster1	79	0.542393	1.30833
cluster2	123	0.462027	0.974767
cluster3	48	1.02655	2.00384
cluster4	92	0.442258	1.1714
cluster5	84	0.742574	1.35946
cluster6	11	0.556278	1.43155

Time taken by KMeans to execute(seconds): 4.9181332

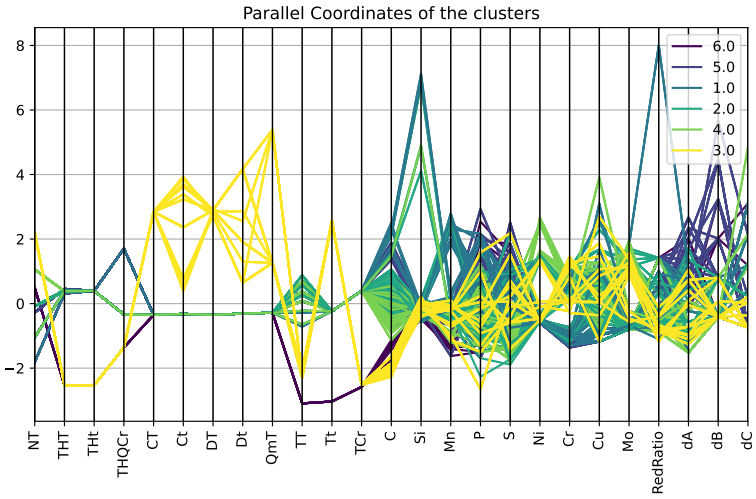


Figure 30: Result summary and parallel coordinates plot of K-means clustering at number of clusters six.

steels are normalized and tempered and some are through hardened and tempered. Some of the other details to note from the result summary. The number of observations on cluster 2(255) is more than the observations on cluster 1(134), the average distance from centroid is lowest for cluster 2 (0.819) and highest for cluster 1 (1.327) and same can be observed in Figure 28. This indicates that cluster 2 has the least variability and cluster 1 has the most variability. To confirm the presence of the above specified steels in the cluster 1 and 2, we once again verify it using parallel coordinate plot.

- In general the low alloy steel can be classified based on their heat treatments, such as nitrided and tempered, quenched and tempered. Similarly spring steels are classified as hardened and tempered. Nitrided steels are in general medium carbon steel that contain strong nitride forming elements like chromium(Cr) and molybdenum(Mo). Other alloying

elements like nickel(Ni), copper(Cu), silicon(Si) and manganese(Mn)also affect nitriding characteristics.

- The contribution of certain hardening features present in Figure 29 confirms the presence of carbon and low alloy steel and spring steel in cluster 1 and cluster 2. One of the heat treatment process of low alloy steel is nitriding and tempering. The chemical compositions of the nitrided steel discussed in the previous point can be observed in cluster 1 and 2. For example, in cluster 1 and 2, when there is contribution from nitriding feature, one can observe that there is also some contribution by tempering features like tempering temperature/time(TT, Tt) and cooling rate for tampering(TCr) and chemical compositions like Cr, Mo, Ni, Cu, Mn and Si to the clusters. Spring steels are through hardened(THT, THt, THQCr) and tempered. The same can be observed in cluster 1, and this also explains the higher variability of cluster 1. This confirms the presence of both carbon and low alloy steel and spring steel in the clusters created using reduced dataset.

- Other important observations from Figure 30. The number of clusters for a given dataset is determine from the elbow plot, in this case for the reduced dataset the number of clusters is three as in Figure 5(b). So when a cluster plot is created with the reduced dataset at number of cluster six as in Figure 7(b) and at cluster 3 as seen in Figure 7(a), the number of observations present in both the clusters remained identical and this is also the case at number of cluster ten. Here, cluster 1 of Figure 7(a) is sub-clustered into 1, 5 and 6 in Figure 7(b) and cluster 2 is sub-clustered into 2 and 4.

- Additional details to note from the the parallel coordinates plot as depicted in Figure 30 for number of cluster six. When creating an parallel plot for number of cluster six the pattern of plotting as seen in the plot legend is similar to that of the parallel plot for number of cluster three(Figure 29). In parallel plot for number of cluster six we have a plot legend of clusters 6, 5, 1, 2, 4 and 3. Here, the clusters 6, 5, 1 refers to the cluster 1 of Figure 29, cluster 2, 4 refers to cluster 2 and cluster 3 is same in both cases thus remain as cluster 3 and plotted last.

- The data assessment by K-means clustering once again confirms that the details present in original dataset is also present in the reduced dataset and the clusters existed in original dataset still exist in extracted dataset or score dataset. The time taken for PCA and K-means to execute at number of cluster three and six are 2.928, 4.918 seconds.

4.2.3 MLR model analysis

After training and validation step the performance of the model is analysed. As mentioned before the MLR model is trained and validated using two combination of loss functions (i.e MSE and RMSE loss) and SGD optimizer. For these two losses there are two different set of hyperparameters as seen in Table 7. Result analysis is done to identify which among the two combinations have higher accuracy and lower loss value for different datasets. To pinpoint which particular function in the programmed model is responsible for the increase in time taken for the model to execute. Here, result analysis is done for four different input datasets as seen in Table 26. These datasets are selected based on their correlation with fatigue strength as seen in Figure 10. Dataset with 12 features refers to the dataset with all positive correlated features. Dataset with 7 features refers to dataset with features that has high degree of correlations(positive) and 3 features refers to dataset with features that has correlations greater than 0.80. From the MLR results present in Table 26 fallowing conclusion can be drawn.

- The implemented MLR model is not over fitting/under fitting it is generalized as the difference between the validation and the training accuracy is very small and the same

Table 26: MLR model training and validation results at both loss combination, at SGD optimizer, and with datasets of different number of independent features.

Dataset	Comb.	Training accuracy	Validation accuracy	Accuracy difference	Training loss	Validation loss	Loss difference
All features	MSE	0.965	0.959	-0.006	0.0339	0.0455	0.0116
	RMSE	0.969	0.957	-0.012	0.1112	0.1447	0.0335
12 features	MSE	0.903	0.896	-0.007	0.0934	0.1159	0.0225
	RMSE	0.902	0.893	-0.009	0.2271	0.261	0.0339
7 features	MSE	0.768	0.79	0.022	0.2244	0.2339	0.0095
	RMSE	0.757	0.778	0.021	0.3327	0.3982	0.0655
3 features	MSE	0.764	0.787	0.023	0.228	0.2374	0.0094
	RMSE	0.76	0.781	0.021	0.3472	0.4011	0.0539

can also be observed in loss difference. In datasets with 7 and 3 features the validation accuracy is greater than the training accuracy. This shows how well the model is generalized as the number of samples used for training a model is 349 where as the samples used to validation is just 88.

- Among all the results, MSE loss at default dataset or all feature dataset has the highest prediction/validation accuracy of 0.959 and the lowest validation loss 0.0455 thus this combination makes the best performing MLR model.
- It can also be seen that in every dataset the model computed with MSE loss has higher accuracy and lower loss than the model computed with RMSE loss. This is because of their mathematical difference as a result RMSE will have very small weight updates during model training.
- At MSE loss the accuracy values decreases from top to bottom and the loss value increases from top to bottom but this is not the case in RMSE as the accuracy of dataset with 3 features is greater than the accuracy of the dataset with 7 features but for loss the behaviour is same as in MSE.
- It is clear that in almost every cases the model performance decreases with decrease in number of features.

The second set of result analysis is done using the comparison excel($resultcomparisionMLR.xlsx$) created with fatigue dataset from which two important details are attained. The mean value of the predicted fatigue strength at MSE and RMSE are 575.72 and 581.23 which is closer to the mean value of the original fatigue strength that is 577.1. This show how close the predicted values are to the target values and the difference between the mean value of target and MSE is just 1.38. From the post processing analysis of the table we where able to observe that RMSE made better prediction of fatigue strength values for 51 samples out of 88 when compared to 37 made by MSE. This contradicts the above statement of MSE being the better model. Here, we can seen that the difference between the R^2 of MSE and RMSE computed at All-features is 0.002 which is considerably low and the reason for this contradiction. But the loss value of MSE is lower than RMSE and the difference in the mean value between RMSE and target is 4.125 which is larger when compared to MSE. This draws a conclusion that, even though RMSE could make more good prediction of fatigue strength, in some cases the prediction is poor. For example, in sample number 6 of the comparison excel the target value is 1056; the

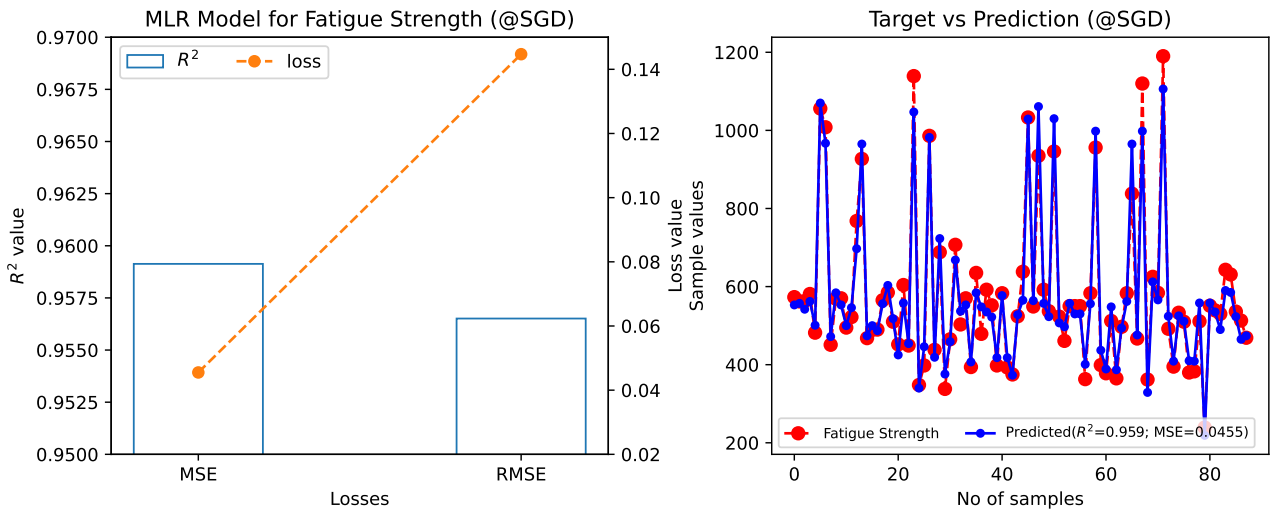


Figure 31: MLR model overall validation results at MSE and RMSE loss, with validation plot of the best performing model to predict the fatigue strength of steel.

value predicted at MSE: 1070 and at RMSE: 1101. The difference between the target and RMSE value is large and the same can be seen in Figure 15(b). So with respect to this analysis it would be safe to say that MSE at All-features dataset produce better model performance than RMSE. Some of the other analysis include the time taken by the MLR model at different instances is given in the Table 27. There are different time metrics present in the table, from amount of time taken by the entire code to execute to time taken for both the combination to execute. One thing that is clear from Table 27 is the time decreases with decrease in number of features and the time taken for the optimizer to run is same for all the datasets. An important observation is that at all instances the time taken to train an MLR model is greater than any other function step present in the model. So in overall the best prediction accuracy of an MLR model in predicting the fatigue strength of steel is 0.959 at MSE loss and 0.957 at RMSE loss and the same is depicted in Figure 31. Selection of MSE or RMSE for prediction completely depends on the input dataset.

Table 27: Time taken in seconds at different instances of the MLR model for dataset with different number of independent features.

Dataset	Comb.	Code	Model	Training	Prediction	Forward Propagation	Backward Propagation	Optimizer	Total time both comb.
All features	MSE	1.57417	0.753365	0.7343	0.00056	0.000005	0.000021	0.000006	3.2240915
	RMSE	1.649916	0.797088	0.779166	0.000833	0.000005	0.000026	0.000006	
12 features	MSE	1.560062	0.709872	0.693389	0.000613	0.000004	0.000018	0.000006	3.1982588
	RMSE	1.638191	0.763921	0.751871	0.00055	0.000004	0.000023	0.000006	
7 features	MSE	1.530182	0.687862	0.672961	0.000542	0.000004	0.000017	0.000006	3.1164856
	RMSE	1.586298	0.75149	0.7368	0.000725	0.000004	0.000022	0.000006	
3 features	MSE	1.511286	0.668749	0.657673	0.00061	0.000003	0.000016	0.000006	3.1006491
	RMSE	1.589356	0.736709	0.724653	0.000471	0.000004	0.000021	0.000006	

4.2.4 ANN hyperparameter tuning

Selecting suitable optimizer: To increase the ANN model performance i.e., to reduce the model loss and to increase the accuracy, four different optimizers are selected. These optimizers include, SGD, SGDMomentum, RMSProp and Adam, more details regarding these optimizers

can be found in the Sects.3.7.5 to 3.7.8. Initially, the model is trained using the standard optimizer inputs and then tuned with respect to the models response to the fatigue dataset. If not tuned properly, the model get stuck in the local minima. The tuned input values can be found in Table 7, and in Table 11 to Table 13. After tuning the optimizers, the optimizer with higher training accuracy(R^2) and lower loss(ls) is considered as the best performing optimizer. For the ANN model with fatigue dataset, among all the optimizers, the training results of the model trained with Adam optimizer has the highest accuracy(0.998) and lowest loss(0.0021), as seen in Figure 32, and the same is selected as the suitable optimizer.

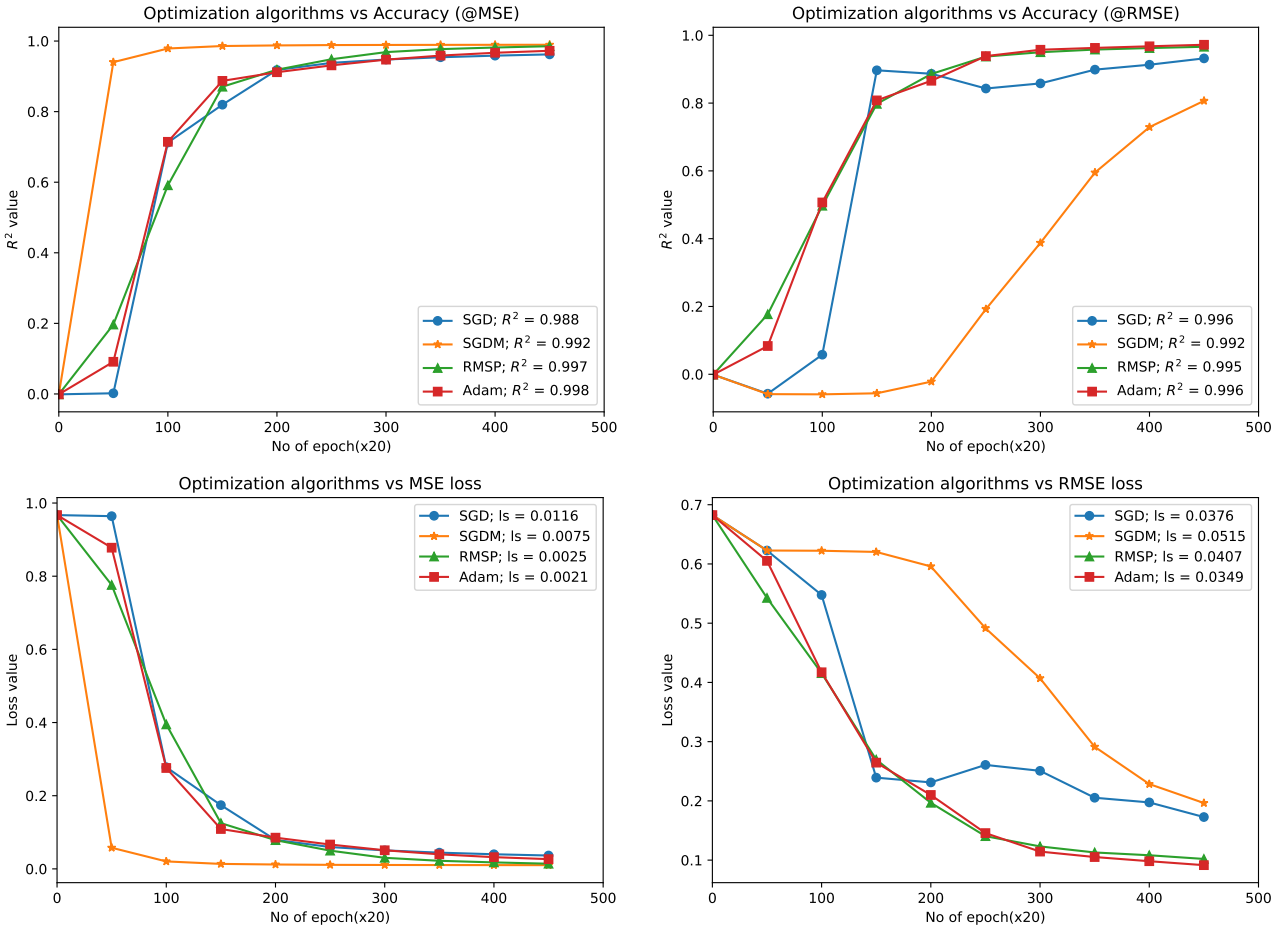


Figure 32: Analysing accuracy(first row) and loss(second row) output of the ANN model during the training step at MSE and RMSE loss for different optimizers.

Selecting suitable number of neurons for hidden layers: The implemented ANN model consist of three layers, two hidden and an output layer. As this is a regression model, the number of neurons(columns) in the output layer is one. The fatigue dataset consist of 25 independent features and same will be the number of neurons in the input layer. The neural network is built in the architecture 25-N1-N2-1, where, N1 and N2 refers to the number of neurons in the hidden layer 1 and 2. The input layer will be modified automatically with respect to the input dataset. The number of neurons in the hidden layer is one of the hyperparameters of neural network that influence the validation loss, accuracy(R^2) and computational time. In the implemented ANN model there are eight combinations of loss and optimizer, in which Adam optimizer at MSE loss is the best performing model with lower loss and higher accuracy and the same can be seen from Figure 35. To select suitable number of neurons that results in lower validation loss and higher accuracy with reasonable computation time, the best performing

model is tuned for four different neuron inputs(70,50,20,10) at N1 and N2, and the same is depicted in Figure 33. From different NN architectures as seen in Figure 33, we can say that, at 25-10-10-1, the model performs the best with higher accuracy and lower computation time. Thus, N1 and N2 at number of neurons 10 is selected for the ANN model and the same is given as an input for *layers* and can be seen in Table 14. Note that it is not necessary to have same number of neurons in both the hidden layers. This tuning is performed in *PPP_ANN_createOverallresults.py*, and for each instance the user can execute the code for any four different number of neurons in both N1 and N2 and in any combination of loss and optimizer. This saves quit a lot of time as we could evaluate the NN model performance for different combinations instead of tuning one combination at a time. All hyperparameter tuning results are created in *PPP_ANN_createOverallresults.py* and the combination plots are made using *PPP_ANN_combination.py*.

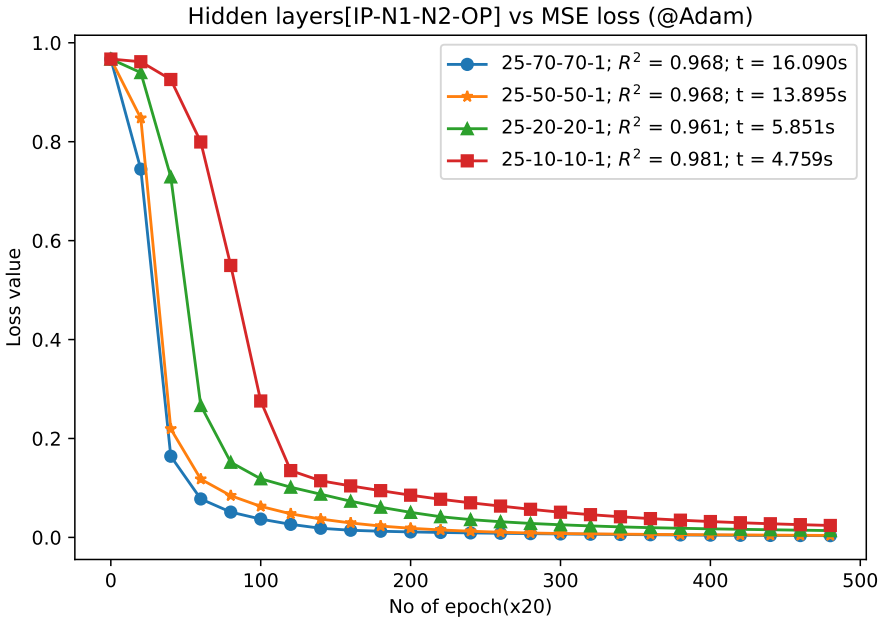


Figure 33: ANN model hyperparameter tuning of hidden layer at different neural network architecture.

Analysing learning rate update: Learning rate is an important parameter as it is responsible for the weightage of parameters-update that occur during the optimization step. Here, we extensively discuss about the relation observed between the learning rate and training loss in SGD and SGDMomentum optimizers. In general, the training loss decreases with decreases in learning rate and the same can be observed in Figure 34. The learning rate(lr) at MSE loss for both optimizers are the same, but different loss responses are observed due to the momentum hyper parameter in SGDM. As discussed in Sect.3.7.6, the SGD optimizer with momentum, accelerates the gradient vector in right direction and leads to faster convergence, and the same can be observed in Figure 34. The hyperparameter, learning rate decay(lr_d), is used to decrease the learning rate over each number of epoch, thus, avoiding fixed learning rate through out the training step. In SGD case, the learning rate decay(lr_d) at MSE(0.1) is greater than the decay in RMSE(0.01). Thus, we can observe a more gradual convergence of training loss with decreasing learning rate at MSE loss than in RMSE loss where the decay rate is small. This shows how sensitive these parameters are during model training. The results of the learning rate and training loss can be found in the file *trainingresults_Loss_Opt.txt* in the directory Results_Training.

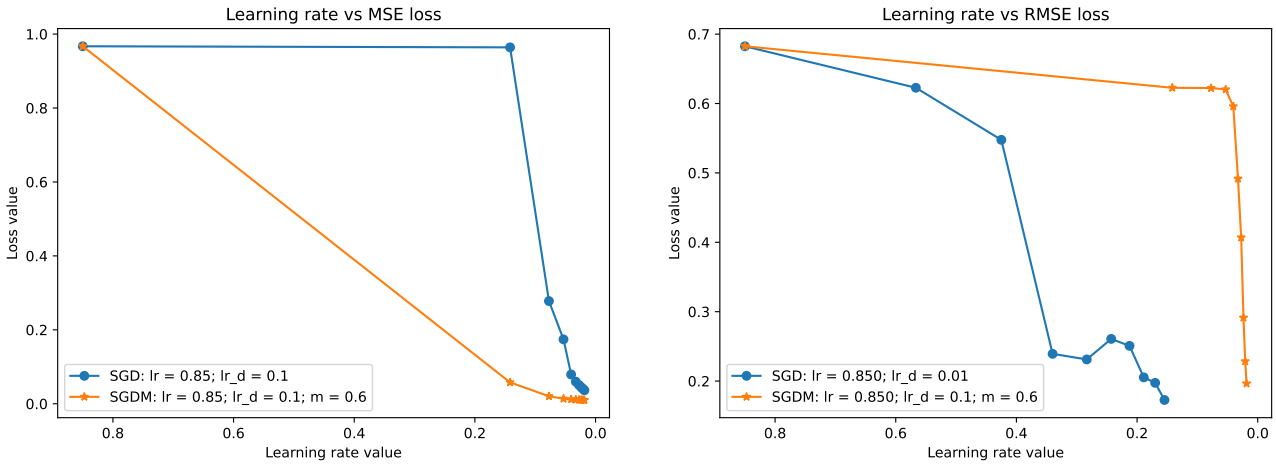


Figure 34: Analysing the relation between learning rate and training loss at MSE and RMSE.

4.2.5 ANN model analysis

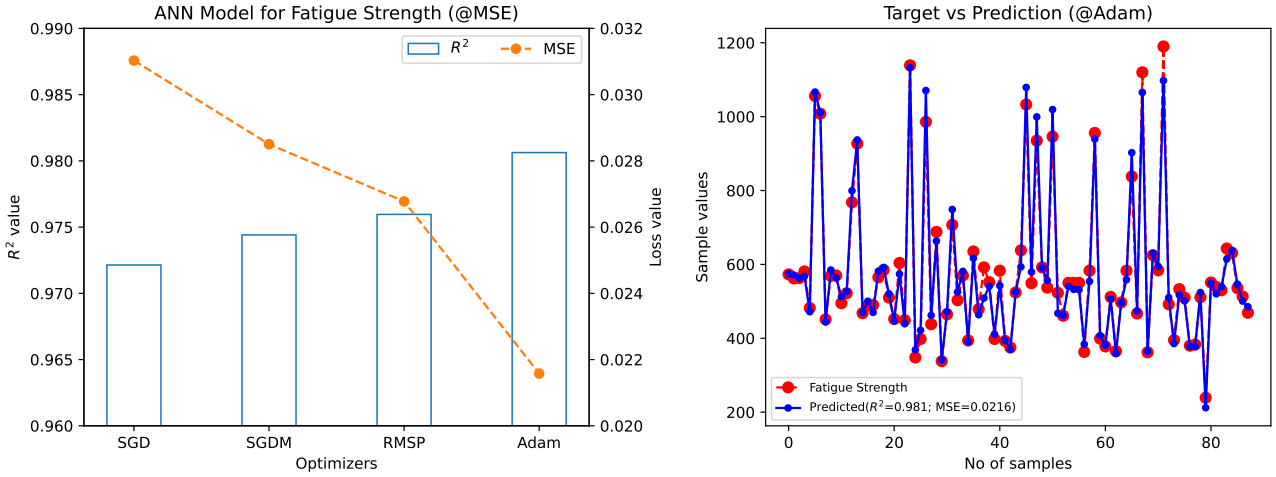


Figure 35: ANN model overall validation results at MSE loss, with validation plot of the best performing model to predict the fatigue strength of steel.

After training and validation step, the performance of the model is analysed. As mentioned before the ANN model is trained and validated for eight combination of loss and optimizers as seen in Figure 36. Result analysis of ANN is done to identify which combination among the eight has higher validation accuracy and lower loss for the fatigue dataset. Unlike in MLR, the result analysis in ANN is done only with the fatigue dataset and not with other feature selected datasets. Additionally, to determine which function in the programmed model is responsible for increase in time taken for the model to execute. From the results generated by the ANN model fallowing conclusions can be made.

- The implemented ANN model is not over fitting/under fitting, it is generalized as the difference between the validation set and the training set for both accuracy and loss(marked with red) is small and the same is observed in Figure 36. The number of samples in the validation set is 88, where as the number of samples in the training set is 349, still the validation results where close enough to the training results.

- The model trained with MSE loss as loss metric and Adam as optimizer shows higher training accuracy(0.998) and lower loss(0.002). Where as the model trained with RMSE

ANN model results for all combinations during training and validation:

Combinations	Training accuracy	Validation accuracy	Accuracy difference	Training loss	Validation loss	Loss difference
MSE_SGD	0.987994	0.972140	-0.015854	0.011597	0.031029	0.019432
MSE_SGDM	0.992267	0.974408	-0.017859	0.007470	0.028503	0.021033
MSE_RMSP	0.997410	0.975959	-0.021451	0.002502	0.026776	0.024274
MSE_Adam	0.997794	0.980627	-0.017167	0.002131	0.021577	0.019446
RMSE_SGD	0.995904	0.969230	-0.026674	0.037627	0.114347	0.076720
RMSE_SGDM	0.992219	0.972876	-0.019343	0.051504	0.116192	0.064688
RMSE_RMSP	0.994924	0.964031	-0.030893	0.040676	0.124938	0.084262
RMSE_Adam	0.995682	0.974694	-0.020988	0.034867	0.114380	0.079512

Figure 36: ANN model training and validation results at all combinations of loss and optimizer.

as loss metric shows higher training accuracy(0.996) and lower loss(0.034) when combined with SGD and Adam optimizer.

- Among all the results, the combination of MSE loss and Adam optimizer is considered to be the best performing combination with the highest validation accuracy of 0.981 and lowest loss of 0.0216 and the same is depicted in Figure 35.
- From Figure 35, we can observe the relation between loss and accuracy. Here, as the loss value decrease there is a gradual increase in accuracy value. The result comparison plot on the left hand side, show that the results predicted by the Adam-MSE combination, almost in all cases matches the target results.

The prediction results of all eight combination are written into a excel file as in MLR, from which certain result analysis are done. The mean value of the predicted fatigue strength for each combinations is computed. Here, the mean of the fatigue strength computed at Adam-MSE combination(577.344) and Adam-RMSE combination(577.807) is closer to the mean of the target results(577.102). The difference between the mean value of the target and the Adam combinations are 0.24 and 0.75. This show how close the predicted results are to the target results. The results attained from the post process analysis of the excel confirms the results attained in Figure 36. The Adam-MSE combination made better prediction of fatigue strength for 18 samples out of 88, where as the Adam-RMSE and SGD-RMSE made better prediction for 14 samples each. Some of the other analysis include the time taken by the ANN model at difference instances is given in the Figure 37. There are different time metrics present in the table, from amount of time taken for each code to execute to time taken by all the eight combinations to execute. Some of the observation made from the Figure 37 are list below.

- Among the optimizers, the SGD optimizer executes the fastest and Adam optimizer takes more time due to fact that it has more robust parameter update step. This optimizer effect can also be observed in time taken by each combination to train.
- The least amount of time taken by a model to execute is the SGD-MSE combination(3.512 seconds). Due to the intense model training(10,000 epoch), the time taken during ANN model training is greater than the time taken by any other function present in the model.

- In overall the prediction accuracy of the ANN model in predicting the fatigue strength of steel is 0.981 at Adam-MSE and 0.975 at Adam-RMSE and the same can be observed in Figure 36. Selection of optimizers and loss metrics completely depends on the input dataset.

Time taken in seconds by ANN at different instances and combinations:

Combinations	Code	Model	Training	Prediction	Forward_prop	Backward_prop	Optimizer
MSE_SGD	3.512751	2.828849	2.349262	0.000504	0.000054	0.000111	0.000013
MSE_SGDM	3.676809	2.968907	2.494906	0.000432	0.000058	0.000113	0.000020
MSE_RMSP	3.904345	3.190775	2.699890	0.000491	0.000063	0.000108	0.000041
MSE_Adam	4.228945	3.991718	3.129532	0.000507	0.000063	0.000120	0.000068
RMSE_SGD	3.784242	4.920857	3.009700	0.000610	0.000072	0.000145	0.000015
RMSE_SGDM	3.884025	3.107880	2.608040	0.000546	0.000058	0.000122	0.000021
RMSE_RMSP	4.064246	3.303440	2.806519	0.000569	0.000061	0.000117	0.000043
RMSE_Adam	4.302717	3.593266	3.101306	0.000577	0.000063	0.000119	0.000067

Time taken by all the eight combination to execute: 31.3580983

Figure 37: Time taken in seconds at different instances of the ANN model for dataset with different number of independent features.

4.2.6 kNN model analysis

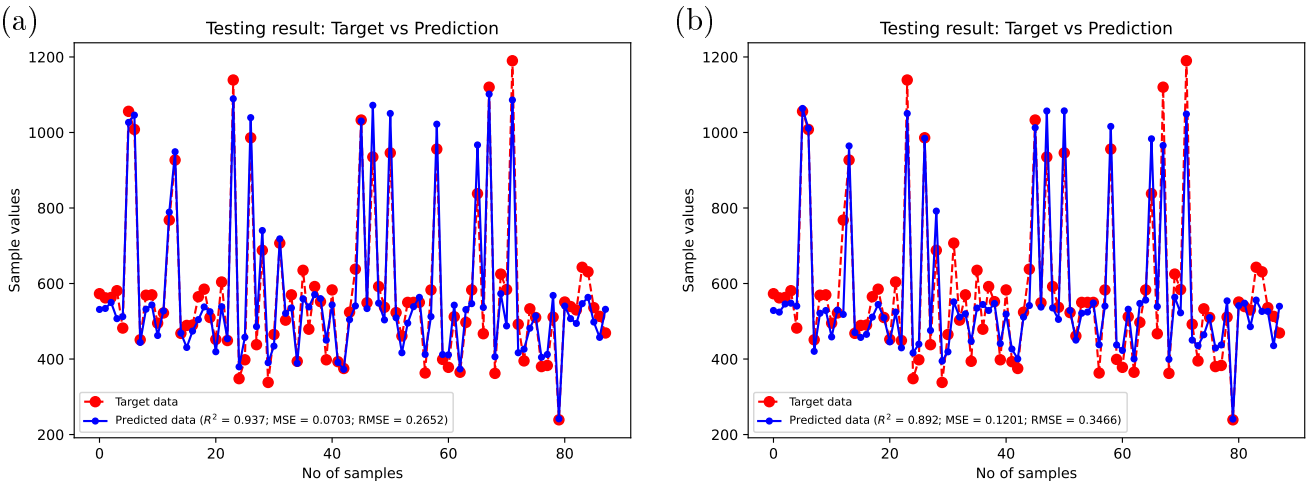


Figure 38: kNN prediction result at MSE loss with (a)Dataset: fatigue dataset;(b)Dataset: all positive features of fatigue dataset.

In kNN the accuracy of the model depends on the selected nearest neighbors(k). The optimal number of neighbors are selected based on lowest model loss for a predefined set of k inputs, and the same can be observed in Figure 22(a). These nearest neighbors are then used to make predictions of the validation set and the output of the same is depicted in Figure 38. Some observation made from the kNN prediction results are listed below.

- The optimal nearest neighbors for fatigue dataset is three, and the kNN model predicted using three neighbors has a prediction accuracy of 0.94 and model loss of 0.0703(at MSE) and the same can be observed from Figure 38(a).

- The time taken by the kNN model to make predictions with the fatigue dataset is lesser when compared to other predictive models implemented in the project and the same can be observed in Figure 22(b).

- The MLR model computed with fatigue dataset has an validation accuracy of 0.96, and the time taken for the model to execute is 1.574. Where as, the kNN model has an accuracy of 0.94, and the time taken for the model to execute is 0.538, which is three times lesser than the time required by an MLR model with a small trade off(0.02) in the accuracy value.

- The optimal nearest neighbor for fatigue dataset with 12 features is nine, and the kNN model predicted using nine neighbors has a prediction accuracy of 0.892 and a model loss of 0.1201(at MSE) and the same can be observed from Figure 38(b). The time taken for the model to execute with the reduced dataset(12 features) is almost same as in the original dataset.

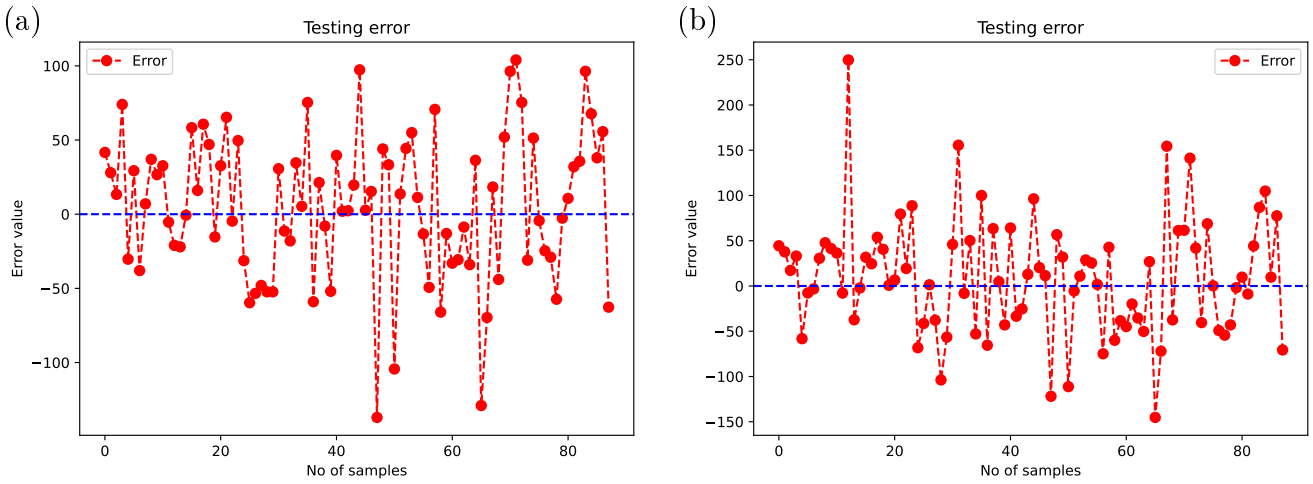


Figure 39: kNN prediction error at MSE loss with (a)Dataset: fatigue dataset;(b)Dataset: all positive features of fatigue dataset.

- An important point to note is that, the kNN model predicted using reduced dataset(12 features) have more observation with accurate predictions(almost zero error) than model predicted using fatigue dataset, the same can be observed(blue line) from Figure 39.

- The kNN regression model is more suitable for sparse and dense dataset with limited dimensions and lower observations. In case of larger dimension, lesser will be the prediction accuracy, and for larger datasets, the models execution time will be high. Thus, kNN model should be preferred with respect to the project dataset.

4.2.7 Overall result analysis and discussion

In this project a framework is devised to predict the fatigue strength of steel. This framework includes, data assessment tools and predictive models. In the data assessment step, the data present in the input dataset is analysed using three different data assessment tools, namely, PCA, K-means and correlation matrix(CMatrix). During predictive modelling, three different ML algorithms, MLR, ANN and kNN are used to predict the target feature. In **PCA**, we determined that 'seven' principal components are required to extract the fatigue dataset. In **K-means**, we identified that 'three' clusters are available for the fatigue dataset. In **CMatrix**,

Table 28: Performance analysis of all the best performing combinations of ANN, MLR and kNN.

Method	Accuracy(R^2)	MSE loss	RMSE loss
ANN	0.980627	0.021577	0.11438
MLR	0.959132	0.045517	0.144713
kNN	0.936856	0.070327	0.265193

we identified that the features such as, 'Tampering time(Tt)', 'Carburization temperature(CT)' and 'Diffusion temperature(DT)' have maximum(0.86-0.85) correlation with fatigue strength. After predictive modelling of fatigue dataset, we observe that among the three ML algorithms, **ANN** made better prediction for fatigue strength with a validation accuracy of 0.981 and a validation loss of 0.021, and the same is observed in Table 28 and Figure 40. The results obtained are with respect to the fatigue dataset, and results may vary for different datasets. This is a general framework that can be applied to datasets that full fills the set of conditions for dataset.

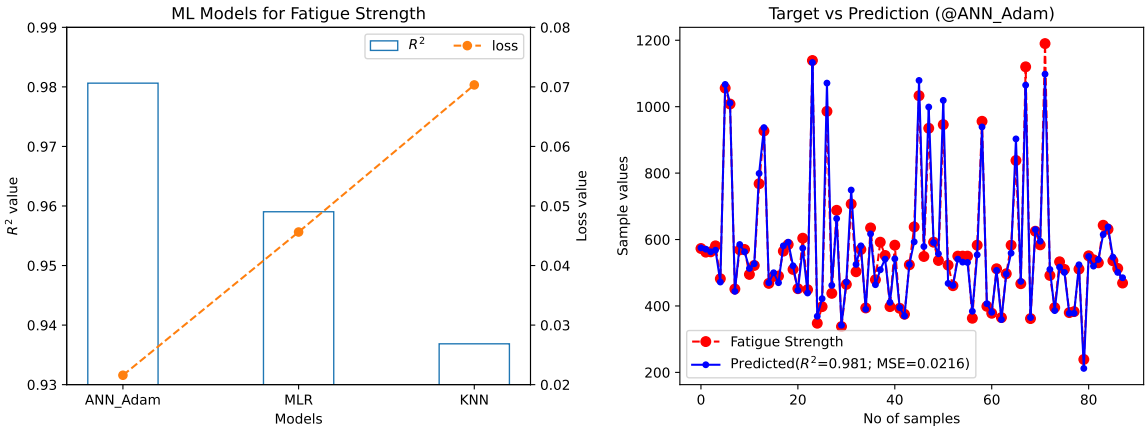


Figure 40: Validation results of predictive modelling with validation plot of the best performing model to predict fatigue strength of steel.

4.3 Concluding remarks

The advanced data analytics and modern statistics tools are the major players for the accelerated and cost effective development of light weight and energy efficient materials that are targeted for current renewable technologies. The important challenge in this approach is to create a reliable linkage between the chemical compositions, processing parameters and the target property. These connections act as a bridge to test different combinations which leads to the development of anti-fatigue high strength steel.

Further development: There is a lot of scope for improvement in this project. As a first step we could implement other regression models like decision trees, random forest and symbolic regression. Combine all implemented models together to form a generalized single regression package. This package can be used to make predictions of the given dataset and create prediction equations based on user input. The vision of this project is to create a sophisticated web-page with all the implemented ML models and data assessment tools to predict mechanical properties like hardness, fracture toughness and tensile strength using pure predictor technique and output structured results for datasets with more than one independent features.

5 Manual

Table 29: Details on files that are executable, files and file-format that are used as inputs, and information regarding how to execute the files with different options and results obtained are specified for each model implemented in this project.

Models Implemented	Executable files	Files used for execution	Execution
PCA and K-means	PPP_KMeans_main.py	Dataset:fatigue_dataset Format: .csv Library:PPP_KMeans Format: .py	python .\PPP_KMeans_main.py - -option Options:Table 4; Results:Table 3
Correlation Matrix	PPP_CMatrix_main.py	Dataset:fatigue_dataset Format: .csv Library:PPP_CMatrix Format: .py	python .\PPP_CMatrix_main.py - -option Options:Table 5; Results:Table 3
Neural Network	PPP_ANN_main.py PPP_ANN_createOverallresults.py PPP_ANN_combination.py	Dataset:fatigue_dataset Format: .csv Library:PPP_ANN Format: .py	python .\PPP_ANN_main.py - -option python .\PPP_ANN_createOverallresults.py - -option python .\PPP_ANN_combination.py - -option Options:Table 14; Results:Table 15
Multi Linear Regression	PPP_MLR_main.py PPP_MLR_createOverallresults.py PPP_MLR_combination.py	Dataset:fatigue_dataset Format: .csv Library:PPP_MLR Format: .py	python .\PPP_MLR_main.py - -option python .\PPP_MLR_createOverallresults.py - -option python .\PPP_MLR_combination.py - -option Options:Table 9; Results:Table 8
k-Nearest Neighbors	PPP_KNN_main.py	Dataset:fatigue_dataset Format: .csv Library:PPP_KNN Format: .py	python .\PPP_KNN_main.py - -option Options:Table 19; Results:Table 17
Overallresults	PPP_Overallresults.py	Resultant files from ANN, MLR and kNN	python .\PPP_Overallresults.py - -option Options:Table 20; Results:Table 18

In this section a detailed overview on how to execute the programmed file with different input options will be discussed with an execution example for each model. For the fatigue dataset used in this project, all the required inputs including the dataset are given as default input values and the same can be observed in 'Options' specified in Table 29. This means, no additional inputs are required for executing the files and to generate the attained model results. This is the case in all the implemented models, and the inputs require modification only in special cases or in the cases when a different dataset is used. The 'execution details' provided in the below section should be considered before executing the models.

5.1 Execution details

- Separate directories must be created for each model except Overallresults as mentioned in Table 29 with all required executables and input files before executing the model.
- Note that not only fatigue_dataset, even other datasets that full fills the set of conditions for dataset can be used as an input in the implemented models.
- All executable files can be executed without any options as each program is assigned with default options and the same can be referred in 'Options' specified in Table 29.
- In case of ANN and MLR certain conditions regarding model execution are discussed in Sect.3.6 and Sect.3.6.10 and the same should be complied.
- As discussed in Sect.3.9, the program *PPP_Overallresults.py* should be executed only after executing all the prediction models. This is due to fact that overall-results uses the resultant files of ANN, MLR and kNN to create results. These files are not given as inputs and are directly accessed by the program. This is the reason for not creating a separate directory for Overallresults and creating directories for other models.

- In some input cases, for example, the 'makeplot' input of the ANN model as seen in Table 14, the input values are restricted to 0 or 1. Here, 0 refers to not perform the specified task and 1 refers to perform the task.
- Make sure that the standard and the external libraries specified in Table 22 are installed and are in the specified versions before executing the programs.

5.2 Execute PCA and K-means

This is an example for how to execute Principal Component Analysis and K-means Clustering. For the fatigue dataset, as mentioned before all the required inputs are assigned as default values and the same can be seen in Table 4. Each model must be executed from its own directory, in this case all executables and input files are stored in the directory **PPP_KMeans**. Here, execution of three cases will be demonstrated as seen in Figure 41. The case1 will be the default case where fatigue_dataset.csv is used and all input values are defined by default in the program(using click) and no inputs are required while executing it from the command line. The case2 is executed with the fatigue_Selecteddataset.csv(dataset with 12 features, created with correlation matrix as discussed in Sect.3.5.2), where all inputs are same as in the default case except the input for *data*. Thus, the *data* must be provided as an input in the command-line while executing the file. The case3 is considered to be the hypothetical case where the program is executed with all inputs just to show the usage of all available options. From Figure 41 fallowing observation can be made.

```
(ml) PS D:\PPP_KMeans> python .\PPP_KMeans_main.py  Case1
(ml) PS D:\PPP_KMeans>
(ml) PS D:\PPP_KMeans> python .\PPP_KMeans_main.py --data fatigue_Selecteddataset.csv
Retry: Clusters are only possible till k = 6. Update kfind to 6                    Case2
(ml) PS D:\PPP_KMeans> python .\PPP_KMeans_main.py --data fatigue_Selecteddataset.csv --kfind 6
(ml) PS D:\PPP_KMeans>
(ml) PS D:\PPP_KMeans> python .\PPP_KMeans_main.py --data dataset.csv --selected_pc 5 --k 4 --kfind 6
--target_column target --plt_size 4 5 4 5          Case3
```

Figure 41: Example execution of PCA and K-means for three different cases.

- There are in total seven lines in Figure 41, in which the first line and the last two lines corresponds to case1 and case3, where as the lines in-between the two cases are the ones that performs case2.
- As specified before, in case1 all options are given by default and no options where used while executing the program.
- The case1 is also called as the project-case, as all required project results for PCA and K-means are generated by executing case1.
- Here, case2 and case3 are only for demonstration purpose. From Table 4 we know that there are in total six input options that can be used to execute the program.
- As specified before, case2 is executed with fatigue_Selecteddataset.csv and the same is observed in line three of Figure 41. This execution resulted in an error suggesting to retry the execution with $kfind$ **6**. The reason behind this error is explained in detail in WCSS computation present in Sect.3.4.2.
- With respect to the error, case2 is re-executed, but this time along with the *data*, $kfind$ is also given as an input and the same is observed in line 5 of Figure 41. An advantage of using click is that it considers the assigned default values if no inputs are given to the options.

- The case3 is used to explain the scenario when a completely new dataset is used for execution. Let us consider the new dataset to be the hardness dataset. Now, the default values that where pre-assigned based on fatigue dataset can not be used for hardness data. Thus, in this case all available options will be given as inputs to execute the program file and the same is observed in the last two lines of Figure 41.

5.2.1 Automatic exit of PCA and K-means

Like in case2 the model automatically exits if certain necessary conditions are not fulfilled. In the implemented PCA and K-means, there are four instances including the instance in case2, for which the program exits automatically with an error message. The details of the four instances are listed below.

- The program exits with an error message: "Error: $selected\_pc$ input must be >= 2", if the input for selected number of principal components is <= 2. The reason for the same is discussed in Sect.3.3.3.
- The program exits with an error message: "Error: $k/kfind$ input must be > 0", if the input for selected number of clusters(k) and endlimit $kfind$ is $<= 0$. The details regarding endlimit and number of clusters(k) are discussed in Sect.3.4.2 and Sect.3.4.3.
- Additionally, the program exits with an error message as seen in case2: "Retry: Clusters are only possible till $k = k - 1$. Update $kfind$ to $k - 1$", if the number of centroids are not equal to the number of clusters.

5.3 Execute CMatrix

This is an example for how to execute Correlation Matrix. For the fatigue dataset all the required inputs are assigned as default values and the same can be seen in Table 5. The required executables and input files are stored in the directory **PPP_CMatrix** from which the CMatrix should be executed. Here, execution of three cases will be demonstrated as seen in Figure 42. The case1 will be the default case with fatigue_dataset.csv, where all inputs are defined by default and no inputs required while executing the program. The case2 is executed with the fatigue_Extracteddataset.csv(extracted data from PCA with number of PC two), where all inputs are same as in the default case except the input for *data*. Similar to 'Execute PCA and K-means' the case3 in CMatrix is also considered to be a hypothetical case where the program is executed with a imaginary fracture toughness dataset. From Figure 42 fallowing observation can be made.

```
(ml) PS D:\PPP_CMatrix> python .\PPP_CMatrix_main.py  Case1
(ml) PS D:\PPP_CMatrix>
(ml) PS D:\PPP_CMatrix> python .\PPP_CMatrix_main.py --data fatigue_Extracteddataset.csv Case2
(ml) PS D:\PPP_CMatrix>
(ml) PS D:\PPP_CMatrix> python .\PPP_CMatrix_main.py --data dataset.csv --target_column fracture
--targetf fracture toughness --selected_features 6    Case3
```

Figure 42: Example execution of CMatrix, at default case with fatigue dataset, at extracted dataset and a imaginary case with fracture dataset.

- There are in total six lines in Figure 42, in which the first and the last two line corresponds to case1 and case3, where as the line three is the one that performs case2.
- As specified, case1 is executed with no options and this is the project-case of CMatrix as all required results for fatigue dataset is generated.

- The case2 is an example for how to execute the CMatrix in case of different dataset(for datasets generated in this project) and the same is observed in line three of Figure 42.
- The case3 is an imaginary case that is used to explain the scenario when a complete new dataset is used for execution. The new dataset considered for this case is a fracture dataset. Thus, certain options as seen in line five and six of Figure 42 are specified as inputs to execute the program file.

5.4 Execute ANN

This is an example for how to execute the implemented Artificial Neural Network prediction model. For fatigue dataset, as specified before all the required inputs are assigned as default values and the same can be referred from Table 14. The required executables and input files of the model are stored in the directory **PPP_ANN**. This is the directory from which the model should be executed. The sample execution of the ANN model with fatigue dataset for three cases can be observed in Figure 43. As seen in Table 29 row three, there are three executable files in ANN. Among these files, the **main** file is used to make predictions of the fatigue dataset for single combination of loss and optimizer and the same is demonstrated in case1. The **createOverallresults** file is used to make predictions for all the eight combinations of loss and optimizer and the execution of the same can be observed in case2. The **combination** file is used to create combination and hyperparameter tuning plots using the results generated from createOverallresults and the execution of the same can be seen in case3. More details on the purpose of these files are discussed extensively in last fifteen line of Sect.3.7. From Figure 43 fallowing observation can be made.

```
(ml) PS D:\PPP_ANN> python .\PPP_ANN_main.py
(ml) PS D:\PPP_ANN> python .\PPP_ANN_main.py --loss RMSE --optimizer Adam          Case1
(ml) PS D:\PPP_ANN> python .\PPP_ANN_main.py --loss RMSE --sgd 0.85 1e-2
(ml) PS D:\PPP_ANN> python .\PPP_ANN_main.py --loss RMSE --optimizer Adam --predictor ON
(ml) PS D:\PPP_ANN>
(ml) PS D:\PPP_ANN> python .\PPP_ANN_createOverallresults.py
(ml) PS D:\PPP_ANN> python .\PPP_ANN_createOverallresults.py --makeplot 1        Case2
(ml) PS D:\PPP_ANN> python .\PPP_ANN_createOverallresults.py --predictor ON
(ml) PS D:\PPP_ANN>
(ml) PS D:\PPP_ANN> python .\PPP_ANN_createOverallresults.py                     Case3
(ml) PS D:\PPP_ANN> python .\PPP_ANN_combination.py
```

Figure 43: Example execution of ANN, at single combination, at all eight combinations and as a pure predictor.

- There are in total eleven lines in Figure 43, in which the first four lines and the last two lines correspond to case1 and case3. The lines in-between case1 and case3 i.e., the lines six, seven and eight performs case2.
- In line one of case1, the main is executed with default inputs that where pre-assigned using click. Some of these default inputs are tuned with respect to fatigue_dataset.csv. We know that the main file generates results for single combination of loss and optimizer and in default case MSE loss and SGD optimizers are used.
- In line two of case1, the main file is executed with a different combination i.e., with RMSE loss and Adam optimizer. This is an example for executing the main file with different combinations and likewise one can execute for eight different combinations of loss and optimizer.

- In line three of case1, the main file is executed with RMSE loss and SGD optimizer(which is by default) with new hyperparameter inputs for SGD optimizer. More details on the hyperparameters of SGD can be found in Sect.3.6.6. This is an example for assigning different hyperparameter inputs with respect to the selected optimizer in a single combination.

- The line four of case1 is an example for executing the main file as a pure predictor at RMSE and Adam optimizer. One can also execute the same for any combinations. For default combination only specifying the *predictor* option will be sufficient. Details on pure predictor can be found in Sect.3.7.9.

- The lines six, seven and eight demonstrates the execution of createOverallresults at *predictor* OFF(default option), at *predictor* OFF and *makeplot*, and at *predictor* ON. The line six generates all project results of ANN except plots for fatigue dataset in a single go. The line seven also generates all the results and in addition creates training, testing and error plots for all the eight combinations using the option *makeplot* 1 and thus called as the project-case for ANN. This program has in total 23 tunable input options, including hyperparameter inputs which are tuned for the fatigue dataset as in Sect.4.2.4 and the same is assigned as default inputs. More details regarding the program can be found in Sect.3.7.10.

- The example execution of case3 is observed in the last two lines of Figure 43. The condition specified in 'Execution details(point four)' should be taken into consideration. The point explains why we execute overall results once again as in line ten before executing the combination file in line eleven.

- In addition to this, one can also execute the programs for new dataset and use different options fallowing the same method discussed in case3 of 'Execute PCA and K-means'.

5.5 Execute MLR

This is an example for how to execute the implemented Multi Linear Regression prediction model. As discussed in the MLR section, an attempt to combine all the prediction models together resulted in coding the MLR model by adapting the program design of the neural network. Thus, the execution procedure of the MLR model will be similar to that of the neural network. For the fatigue dataset all the inputs required are assigned by default and the same can be referred from Table 9. Like in other cases the executable files of MLR model must be executed from its own directory. In this case, all executables and input files are stored in the directory **PPP_MLR**. The execution of MLR model with fatigue dataset for three cases can be observed in Figure 44. Similar to ANN, the MLR also has three executables with only slight changes in the purpose of the files. The purpose of the **main** file is similar to that of the ANN and same is executed in case1. The **createOverallresults** file is used to make predictions for two combinations of losses and SGD optimizer and the same can be observed in case2. The **combination** file creates overall results for two loss cases and the execution of the same can be seen in case3. Additional details on the purpose of the files are discussed in the last ten lines of Sect.3.6. From Figure 44 fallowing observation can be made.

- There are in total ten lines in Figure 44, in which the first four lines and the last two lines corresponds to case1 and case3, where as lines six and seven executes case2.

- The line one of case1 is executed with pre-assigned default inputs which are tuned using fatigue_dataset.csv. The execution of main file generates results for the combination of MSE loss and SGD optimizer.

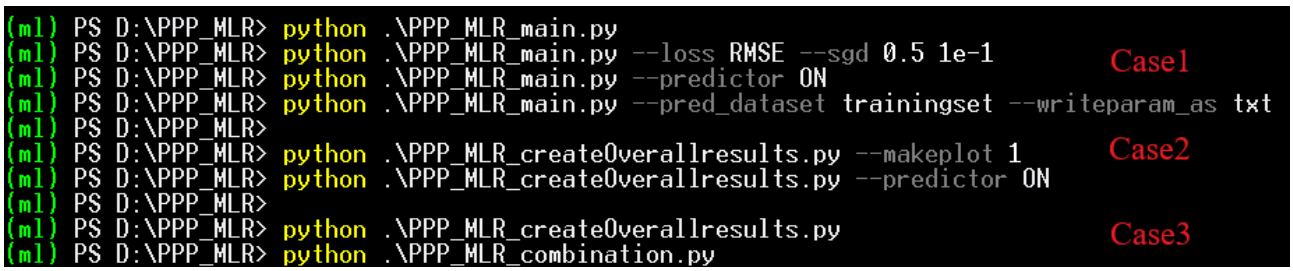


Figure 44: Example execution of MLR, at single combination, at two combinations and as a pure predictor.

- The line two of case2 is an example for executing the main file with RMSE loss and different hyperparameter for the SGD optimizer. For details on hyperparameters of SGD optimizer refer Sect.3.6.6.
- The line three of case1 is an example for executing the main file as a pure predictor with default loss function(MSE loss). Details on pure predictor for MLR can be found in Sect.3.6.10.
- The line four of case1 is for demonstration purpose, where training-set is used as the prediction dataset and the updated parameters are written in .txt format. Training sets are used to check the correctness of the implementation by checking for model Overfitting. More details regarding the same shall be referred from Sect.4.1.3. By default the updated parameters are written in .npy format, as the program reads-in the written file expects .npy format. Details regarding the written format are discussed in the last seven lines of the Sect.3.6.8.
- The line six in Figure 44 is the project-case for the MLR model as it generates all results for two combination of losses and SGD optimizer. The line seven executes as a pure predictor and generates predictor results for both loss combinations.
- The execution of line nine and ten is same as discussed in case3(point seven) of 'Execute ANN'. Similarly, different options can be used if the model is executed using new dataset.

5.5.1 Automatic exit of ANN and MLR

As discussed in Sect.5.2.1, the implemented ANN and MLR programs also exits automatically by suggesting possible inputs, if the given input values are not within the specified range or not fulfilling the input conditions. In MLR, there are six instances at which the program exits. In ANN, the program exits for nine instances. These instances include, *pred_dataset*, *loss*, *makeplot*, *writeparam_as*, *writehl*, *predictor*, *no_of_epoch*, *layers* and *optimizer*. Here, *writehl*, *layers* and *optimizer* are the instances of ANN alone and are not a part of MLR. These are input options that can be found in Table 9 and Table 14. The nine instances of ANN and their checking procedure implementations along with their exit conditions and error messages can be observed(numbered in red) in Figure 45. This is also the same in case of MLR but with only six instances.

5.6 Execute kNN

This is an example for how to execute the implemented k-Nearest Neighbors prediction model. Like in other models the parameters of the kNN model are also tuned for fatigue dataset and are assigned as default values and the same can be seen in Table 19. All required executable

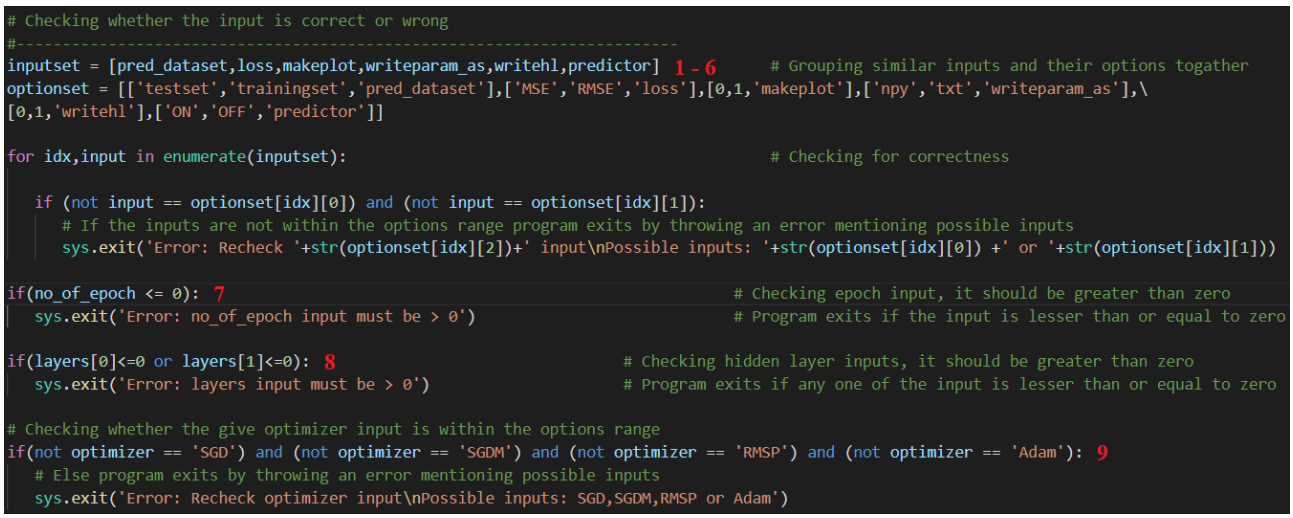


Figure 45: The automatic exit conditions and their error messages for the ANN programs, if certain user input values are not within the offered range or not fulfilling the input condition.

and input files of the model are stored in the directory **PPP_KNN** from which the model should be executed. Here, execution of two cases are demonstrated and the same can be seen in Figure 46. The case1 is the default case where fatigue_dataset.csv is used and all inputs are defined by default and no input is required during execution. The case2 is executed with fatigue_Selecteddataset.csv, where *data* and k(nearest neighbors) are given as inputs in the command-line while executing the file. From the execution and outputs of the kNN model seen in Figure 46 fallowing observations can be made.

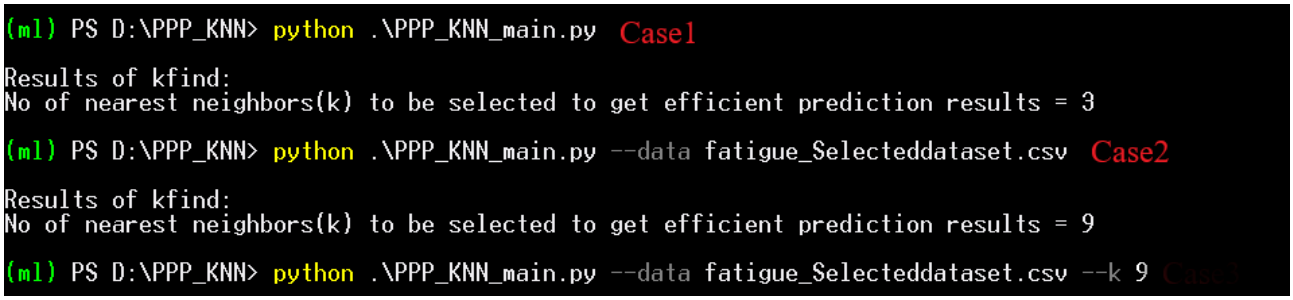


Figure 46: Example execution of kNN and its results for finding the nearest neighbors(k) for two datasets.

- **The idea behind the kNN execution:** While executing the kNN model for a given dataset, one is not aware of what should be the input value for number of nearest neighbors(k). So, the first execution will be based on random value that was assigned by default. Thus, the results generated during the first execution are not accurate predictions. The terminal output of the first execution will result in number of nearest neighbors(k) to be selected for a given dataset. Now the program should be re-executed based on the resultant k value to get efficient prediction results and the same can be observed in Figure 46.
- There are in total three executions and two results in Figure 46, in which first execution and result present in first four lines corresponds to case1 and the remaining executions and result refers to case2.
- The first execution as seen in line one is executed with fatigue dataset. For this case, based on the results, the number of nearest neighbors(k) to be selected for better prediction

is three and same is assigned as an input by default and can be seen in the last row of Table 19. This case is called as the project-case as all the kNN model results are generated executing this case.

- The second execution as seen in line six is an example case in which the kNN model is executed with fatigue_Selecteddataset.csv. For this case, the results generated as in line eight shows that the k to be selected for efficient prediction is nine. This means the result generated in this case with default k, which is three, does not make good predictions.
- As a result the model is re-executed for fatigue_Selecteddataset.csv at $k = 9$ and the same is observed in the last line of Figure 46. This two step execution process is an example for how to execute the kNN model for new datasets.

5.6.1 Automatic exit of kNN

As discussed in the other models, kNN also exits automatically with an error message in three instances when certain necessary conditions are not fulfilled. The details of the instances are listed below.

- The program exits with an error message: "Error: $k/kfind$ input must be > 0", if the input k and endlimit $kfind$ is $<= 0$. The details regarding k and endlimit are discussed in Sect.3.8.2.
- The program exits with an error message: "Error: Recheck *loss* input. Possible inputs: MSE or RMSE", when some other input is given instead of what is specified in the loss option help as seen in second row last column of Table 19.

5.7 Execute Overall-results

This is an example for how to execute the Overallresults program after executing all the prediction models. More details on Overallresults can be found in Sect.3.9. Here, execution of three cases are demonstrated and the same can be seen in Figure 47. The case1 is the default case where loss functions and optimizer are selected based on the observations discussed in Sect.3.9. In this case no inputs are required while executing it in the command line. Case2 explain how to execute the program if in case a different optimizer should be used. Case3 is an hypothetical case where an imaginary fracture dataset is used to explain the usage of other input options. Here, the Overallresults are executed from the parent directory where the directories of all prediction models exists. From Figure 47 fallowing observations can be made.

```
(ml) PS D:\Program> python .\PPP_Overallresults.py Case1
(ml) PS D:\Program>
(ml) PS D:\Program> python .\PPP_Overallresults.py --models ANN_SGDM MLR KNN --optimizer SGDM 2
(ml) PS D:\Program>
(ml) PS D:\Program> python .\PPP_Overallresults.py --len_test 100 --targetf fracture toughness 3
```

Figure 47: Example execution of Overallresults at default case, at different optimizer case and a imaginary case with fracture dataset.

- There are in total five lines in Figure 47, in which the first, third and fifth lines corresponds to case1, 2 and 3.
- The case1 is the project-case where overall results of the prediction models for the fatigue dataset is generated. In this case, all input options are pre-assigned based on the results of all prediction models.

- The case2 is an example case for how to execute the Overallresults program if a different *optimizer* is used. As seen in line three of Figure 47, the overallresults is executed with SGDM optimizer.
- The case3 is an imaginary case which is used to demonstrate the important changes on the inputs in case of a new dataset. Here, a fracture dataset with hundred test samples is considered as an example and other inputs are assigned with their default values.

Literature

[1] A Agrawal et al. "Exploration of data science techniques to predict fatigue strength of steel from composition and processing parameters". In: *Procedia Engineering* 3(1) (2014), pp. 90–108. ISSN: 2193-9772. DOI: 10.1186/2193-9772-3-8.

[2] Jie Xiong, TongYi Zhang, and SanQiang Shi. "Machine learning of mechanical properties of steels". In: *Science China Technological Sciences* 63(7) (2020), pp. 1247–1255. ISSN: 1869-1900. DOI: 10.1007/s11431-020-1599-5.

[3] K. Hartmann, J. Krois, and B. Waske. *E-Learning Project SOGA. Department of Earth Sciences, Freie Universitaet Berlin*. 2018. URL: https://www.geo.fu-berlin.de/en/v/soga/Geodata-analysis/Principal-Component-Analysis/index.html (visited on 02/07/2022).

[4] Saed Sayad. *K-Means*. 2022. URL: https://www.saedsayad.com/clustering_kmeans.htm (visited on 01/29/2022).

[5] Ruben Geert van den Berg. *Pearson Correlation Coefficient - Quick Introduction*. 2016. URL: https://www.spss-tutorials.com/pearson-correlation-coefficient/ (visited on 01/28/2022).

[6] Statistics Solutions. *Pearson's Correlation Coefficient - Statistics Solutions*. 2022. URL: https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/pearsons-correlation-coefficient/ (visited on 01/28/2022).

[7] Adam Hayes. *Multiple Linear Regression (MLR) Definition*. 2022. URL: https://www.investopedia.com/terms/m/mlr.asp (visited on 02/02/2022).

[8] D Chicco, MJ Warrens, and G Jurman. "The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation". In: *PeerJ. Computer science* (2021). ISSN: 2376-5992. DOI: 10.7717/peerj-cs.623.

[9] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. URL: https://ruder.io/optimizing-gradient-descent/ (visited on 02/05/2022).

[10] Abdulkadir Cevik and Ibrahim H. Guzelbey. "Neural network modeling of strength enhancement for CFRP confined concrete cylinders". In: *Building and Environment* 43(5) (2008), pp. 751–763. ISSN: 0360-1323. DOI: 10.1016/j.buildenv.2007.01.036.

[11] Jason Brownlee. *How to Choose an Activation Function for Deep Learning*. 2021. URL: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/ (visited on 02/03/2022).

[12] Raimi Karim. *Intuitions on L1 and L2 Regularisation*. 2018. URL: https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261 (visited on 02/04/2022).

[13] Tao Yao. *Introduction to K-means Clustering and its Application*. 2021. URL: https://medium.com/web-mining-is688-spring-2021/introduction-to-k-means-clustering-and-its-application-in-customer-shopping-dataset-656dcb0a5d09 (visited on 02/26/2022).

[14] UFLDL Stanford. *Debugging: Gradient Checking*. 2022. URL: http://ufldl.stanford.edu/tutorial/supervised/DebuggingGradientChecking/ (visited on 02/10/2022).

[15] Llewelyn Fernandes. *Analyze the Results of a K-means Clustering*. 2022. URL: https://openclassrooms.com/en/courses/5869986-perform-an-exploratory-data-analysis/6177861-analyze-the-results-of-a-k-means-clustering (visited on 02/23/2022).