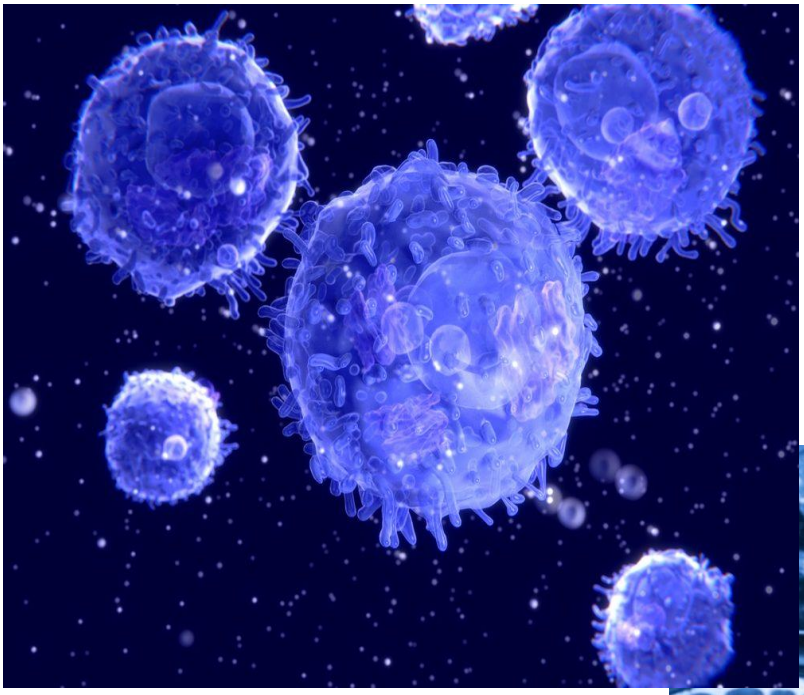
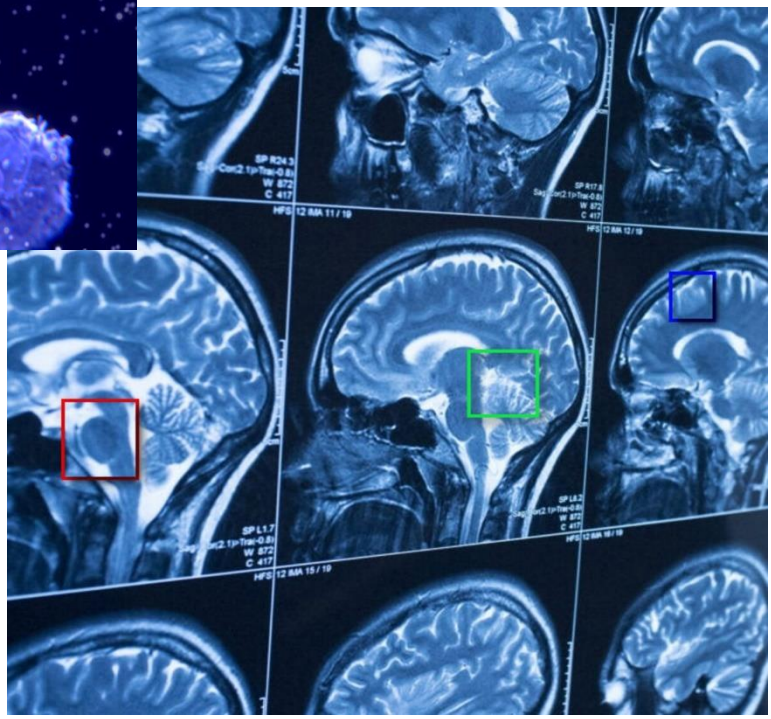
Data Tools Final Project Report





- Lymphoma Cancer dataset
- brain tumor detection with image dataset

Lymphoma Cancer Dataset

Libraries we used:

1. pandas (import pandas as pd): Pandas is a powerful data manipulation and analysis library. It provides data structures and functions for efficiently handling structured data, such as CSV files or database tables. In your project, pandas is likely used for loading and preprocessing the dataset, as well as performing data exploration and manipulation tasks.

2. numpy (import numpy as np): NumPy is a fundamental library for scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. In your project, numpy is likely used for numerical computations and array operations.

3. matplotlib.pyplot (import matplotlib.pyplot as plt): Matplotlib is a popular plotting library in Python. The pyplot module provides a simple interface for creating various types of plots, such as line plots, bar plots, scatter plots, and histograms. In your project, matplotlib.pyplot is used for visualizing data and creating plots to analyze patterns and relationships.

4. seaborn (import seaborn as sns): Seaborn is a statistical data visualization library built on top of matplotlib. It provides a high-level interface for creating attractive and informative statistical graphics. Seaborn simplifies the process of creating visualizations and enhances the visual appeal of plots. In your project, seaborn is likely used for creating visually appealing and informative plots for data exploration and analysis.

5. plotly.express (import plotly.express as px): Plotly is an interactive visualization library that offers a wide range of interactive plots, charts, and maps. Plotly Express is a high-level interface to Plotly, making it easy to create interactive visualizations with minimal code.

6. warnings: The warnings module is used to control the display of warning messages in Python. By using warnings.filterwarnings('ignore'), you are temporarily suppressing the display of warning messages. This can be useful to avoid cluttering the output when running your code.

7. sklearn.metrics: The sklearn.metrics module provides various evaluation metrics for assessing the performance of machine learning models. In your project, the imported metrics (confusion_matrix, accuracy_score, classification_report) are likely used to evaluate the performance of the classification models you trained.

8. imblearn.over_sampling.SMOTE: SMOTE (Synthetic Minority Over-sampling Technique) is a technique used to address class imbalance in datasets. The SMOTE class from the imblearn.over_sampling module is used to oversample the minority class by creating synthetic samples. In your project, SMOTE is likely used to address the class imbalance issue in the lymphoma cancer dataset.

9. sklearn.model_selection: The sklearn.model_selection module provides functions and classes for model selection and evaluation. In your project, the imported modules (cross_val_score, KFold, GridSearchCV, RandomizedSearchCV, train_test_split) are likely used for tasks like cross-validation, hyperparameter tuning, and splitting the dataset into training and testing sets.

10. sklearn.feature_selection: The sklearn.feature_selection module provides classes and functions for feature selection in machine learning. In your project, the imported modules (SelectKBest, mutual_info_classif, RFE, VarianceThreshold, SequentialFeatureSelector) are likely used to perform feature selection techniques and select the most relevant features for the classification task.

11. sklearn.decomposition: The sklearn.decomposition module provides classes for performing dimensionality reduction techniques, such as Principal Component Analysis (PCA). In your project, PCA is likely used for reducing the dimensionality of the dataset and extracting important features.

12. sklearn.random_projection.GaussianRandomProjection: Gaussian Random Projection is a dimensionality reduction technique that projects high-dimensional data onto a lower-dimensional subspace. The GaussianRandomProjection class from the sklearn.random_projection module is used to perform this projection. In your project, it is likely used as an alternative dimensionality reduction technique.

13. scipy.stats.uniform, scipy.stats.randint: These functions from the scipy.stats module are used to define uniform and discrete (integer) probability distributions. They are commonly used in hyperparameter tuning, where you specify a range of values to sample from during the search for optimal hyperparameters.

14. sklearn.neural_network.MLPClassifier: MLPClassifier is a class from the sklearn.neural_network module that implements a multi-layer perceptron (MLP) classifier, which is a type of artificial neural network. In your project, MLPClassifier is likely used as one of the classification models to train and evaluate.

15. sklearn.neighbors.KNeighborsClassifier: KNeighborsClassifier is aclass from the sklearn.neighbors module that implements the k-nearest neighbors algorithm for classification. In your project, KNeighborsClassifier is likely used as one of the classification models to train and evaluate.

16. sklearn.ensemble.RandomForestClassifier: RandomForestClassifier is a class from the sklearn.ensemble module that implements the random forest algorithm for classification. In your project, RandomForestClassifier is likely used as one of the classification models to train and evaluate.

17. sklearn.svm.SVC: SVC (Support Vector Classifier) is a class from the sklearn.svm module that implements the support vector machine algorithm for classification. In your project, SVC is likely used as one of the classification models to train and evaluate.

18. sklearn.tree.plot_tree: The plot_tree function from the sklearn.tree module is used to visualize decision trees. In your project, this function is likely used to plot the decision tree model and gain insights into the tree structure.

19. KMeans from sklearn.cluster: KMeans is a popular clustering algorithm that partitions a dataset into K clusters, where each data point belongs to the cluster with the nearest mean.

20. DBSCAN from sklearn.cluster: DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a clustering algorithm that groups together data points that are close to each other and marks data points as outliers that are in low-density regions.

21. AgglomerativeClustering from sklearn.cluster: Agglomerative clustering is a hierarchical clustering algorithm that recursively merges the nearest pairs of clusters until only a single cluster remains.

22. silhouette_score from sklearn.metrics: Silhouette score is a metric used to calculate the goodness of a clustering technique. It measures how well-defined the clusters are in a given clustering.

23. dendrogram, linkage from scipy.cluster.hierarchy: These are functions related to hierarchical clustering. linkage is used to compute the hierarchical clustering of a dataset, and dendrogram is used to visualize the clustering hierarchy as a dendrogram.

24. NearestNeighbors from sklearn.neighbors: Nearest Neighbors is a class for unsupervised neighbors-based learning. In the context of clustering, it can be used for finding the nearest neighbors of a point, which is useful for algorithms like DBSCAN that rely on the density of points.

Code in Jupyter :

In [17]:

```
import pandas as pd
import numpy as np
import io
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import warnings
warnings.filterwarnings('ignore')
plt.rcParams['figure.figsize'] = (16.0, 9.0)

from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report

from imblearn.over_sampling import SMOTE

from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import train_test_split

from sklearn.feature_selection import SelectKBest, mutual_info_classif
from sklearn.feature_selection import RFE
from sklearn.feature_selection import VarianceThreshold
from mlxtend.feature_selection import SequentialFeatureSelector

from sklearn.decomposition import PCA
from sklearn.random_projection import  GaussianRandomProjection
from scipy.stats import uniform, randint

from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.tree import plot_tree
```

```
from sklearn.cluster import KMeans
from sklearn.cluster import DBSCAN
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import silhouette_score
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.neighbors import NearestNeighbors
```

Data Exploration

Read Data : B-cell Lymphoma is a microarray dataset consists of 3 class of types of cancer

- ☆ DLBCL (Diffuse Large B-Cell Lymphoma): This is considered an aggressive type of lymphoma. However, it is also one of the most common types and often responds well to chemotherapy. With appropriate treatment, a significant percentage of patients with DLBCL can achieve remission or cure.
- ☆ FL (Follicular Lymphoma): Follicular lymphoma is usually indolent or slow-growing. While it is not considered curable with current treatments, many patients can live for many years with this condition. Treatments can control the disease, and some individuals may have long periods of remission .
- ☆ CLL (Chronic Lymphocytic Leukemia): CLL is a type of leukemia rather than a lymphoma, but it is closely related and affects similar white blood cells. Survival rates for CLL can vary widely, and some patients may not require immediate treatment upon diagnosis. There are various treatment options available, and the disease progression can be slow in many cases.

Code and Output:

n [5]:
```
1 Lymphoma=pd.read_csv('Lymphoma.csv')
2 Lymphoma.head()
```

ut[5]:

E1931X	GENE1930X	GENE3129X	GENE3126X	...	GENE3931X	GENE2588X	GENE3120X	GENE6X	GENE5X	GENE3X	GENE2X	GENE48X	GENE47X	class
-0.02	-0.57	-0.17	-0.25	...	0.40	0.02	0.79	0.64	0.16	1.22	1.37	-0.04	0.16	b'DLBCL'
-0.05	-0.38	-0.55	0.35	...	0.57	0.52	-0.23	0.30	0.09	-0.20	-0.05	-0.14	-1.15	b'DLBCL'
-1.13	-0.89	-0.49	-0.23	...	1.62	-0.01	NaN	0.29	-0.57	1.20	1.40	0.29	0.25	b'DLBCL'
-0.65	-0.82	-0.30	-0.22	...	0.34	0.02	0.08	0.49	0.29	1.26	1.24	0.05	0.70	b'DLBCL'
0.06	-0.15	-0.61	-0.65	...	0.16	0.66	1.11	0.28	0.12	-0.16	-0.72	-0.04	-0.22	b'DLBCL'

- We use the info() function to get a summary of the dataset, including the number of rows, columns, data types, and missing values.
- And the describe() function to obtain descriptive statistics such as count, mean, standard deviation, minimum, maximum, and quartiles for numerical columns.

Code and output :

In [4]:
```
1 Lymphoma.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 66 entries, 0 to 65
Columns: 4027 entries, GENE1835X to class
dtypes: float64(4026), object(1)
memory usage: 2.0+ MB
```

In [7]:
```
1 Lymphoma.describe().T
```

Out[7]:

	count	mean	std	min	25%	50%	75%	max
GENE1835X	64.0	-0.105469	0.450960	-1.27	-0.3300	0.010	0.1775	0.81
GENE1836X	56.0	-0.111071	0.428764	-1.49	-0.3925	-0.015	0.1625	0.70
GENE1865X	64.0	0.027344	0.454645	-1.12	-0.2475	-0.015	0.3375	0.87
GENE1380X	59.0	-0.017627	0.400079	-0.80	-0.2850	-0.060	0.2850	1.07
GENE1933X	59.0	-0.006102	0.382424	-0.96	-0.2200	-0.020	0.1700	1.12
...	...	...	...	...	...	...	...	...
GENE5X	66.0	-0.018939	0.374031	-0.89	-0.2500	0.010	0.1750	0.79
GENE3X	61.0	0.008852	0.920053	-2.61	-0.5600	0.000	0.5700	2.16
GENE2X	62.0	0.037419	0.850055	-1.72	-0.5500	0.030	0.5700	3.09
GENE48X	59.0	0.020508	0.341893	-0.94	-0.2150	0.060	0.2700	0.64
GENE47X	60.0	0.019500	0.701435	-1.71	-0.5300	0.030	0.5700	1.39

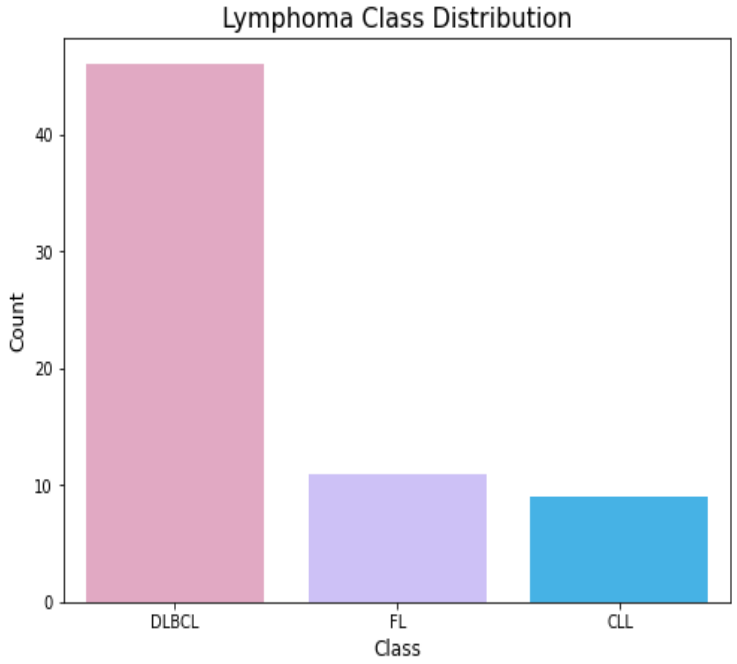
4026 rows × 8 columns

We use bar plot to show count of 3 class of types of cancer :

- DLBCL (Diffuse Large B-Cell Lymphoma)
- FL (Follicular Lymphoma)
- CLL (Chronic Lymphocytic Leukemia)

Code and output :

In [8]:
```
colors = ['#EAA0C2', '#C8B8FF', '#2CBCFF']
# Calculate the count of each class
class_counts = Lymphoma['class'].value_counts()
order = class_counts.index.tolist()
plt.figure(figsize=(8, 6))
sns.countplot(data=Lymphoma, x='class', palette=colors, order=order)
plt.title('Lymphoma Class Distribution', fontsize=16)
plt.xlabel('Class', fontsize=12)
plt.ylabel('Count', fontsize=12)
# Assigning meaningful labels to the classes (DLBCL=0 ,FL=1, CLL=2)
plt.xticks([0, 1, 2], ['DLBCL', 'FL', 'CLL'])
plt.show()
```



And we conclude that a large number of patients have Diffuse Large B-Cell Lymphoma Cancer unlike Chronic Lymphocytic Leukemia

Data Preprocessing

- Print sum of missing data in each column
- Use pandas to handle missing values by using functions like fillna() to identify and handle missing data appropriately with mean value.

Code :

In [9]:
```
# there are genes missing
Lymphoma.isna().sum()
```

Out[9]:
```
GENE1835X     2
GENE1836X    10
GENE1865X     2
GENE1380X     7
GENE1933X     7
             ..
GENE3X        5
GENE2X        4
GENE48X       7
GENE47X       6
class         0
Length: 4027, dtype: int64
```

In [10]:
```
# replace genes expression with mean
numeric_columns = Lymphoma.select_dtypes(include=['number']).columns
numeric_columns_to_fill = [col for col in numeric_columns if col != 'class']
Lymphoma[numeric_columns_to_fill] = Lymphoma[numeric_columns_to_fill].fillna(Lymphoma[numeric_columns_to_fill].mean())
```

- Print sum of duplicated rows
- Rename classes to resample it after that

Code:

In [11]:
```
# check for dublicated data
duplicated_rows = Lymphoma.duplicated()
print("Number of duplicated rows:", duplicated_rows.sum())
```

```
Number of duplicated rows: 0
```

In [12]:
```
# rename classes
Lymphoma['class'] = Lymphoma['class'].str.replace("b'", "").str.replace("'", "")
# Check unique values to confirm
print(Lymphoma['class'].unique())
print('Lymphoma shape',Lymphoma.shape)
```

```
['DLBCL' 'FL' 'CLL']
Lymphoma shape (66, 4027)
```

Resampling classes:

- apply the SMOTE algorithm to oversample the minority class, generating synthetic examples.
- update the dataset with the oversampled data.
- calculate the class distribution after oversampling and prints it.

Code :

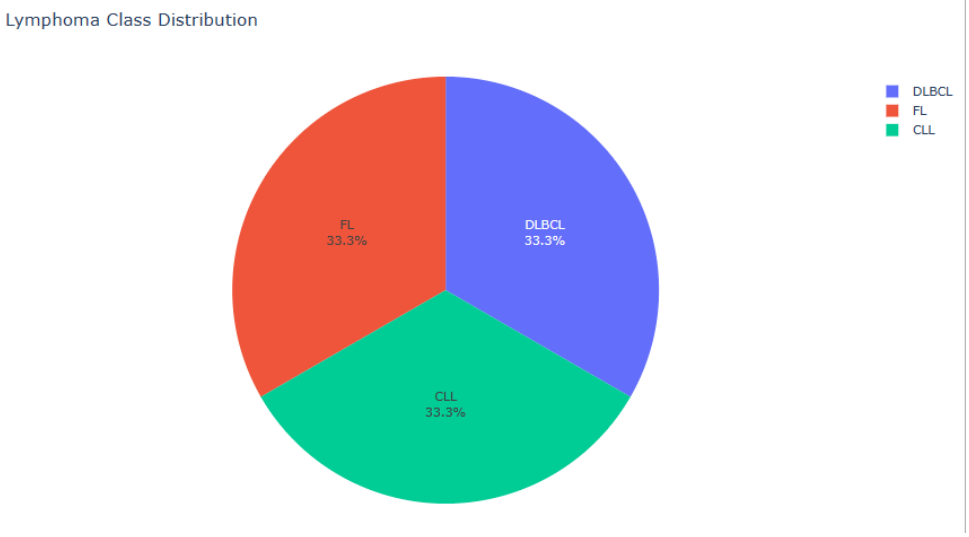
In [18]:

```
# Calculate class distribution before oversampling
original_class_distribution = Lymphoma['class'].value_counts(normalize=True) * 100
print("Original Class Distribution:")
print(original_class_distribution)

# Separate features and labels
X = Lymphoma.drop('class', axis=1)  # Features
y = Lymphoma['class']  # Labels

# Instantiate SMOTE to oversample the minority class within the DataFrame
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)

# Update the DataFrame with the resampled data
Lymphoma = pd.concat([X_resampled, y_resampled], axis=1)

# Calculate class distribution after oversampling
new_class_distribution = Lymphoma['class'].value_counts(normalize=True) * 100
print("\nClass Distribution after SMOTE Oversampling:")
print(new_class_distribution)
print('Lymphoma shape after resampling',Lymphoma.shape)
```

```
Original Class Distribution:
DLBCL    69.696970
CLL      16.666667
FL       13.636364
Name: class, dtype: float64

Class Distribution after SMOTE Oversampling:
DLBCL    33.333333
FL       33.333333
CLL      33.333333
Name: class, dtype: float64
Lymphoma shape after resampling (138, 4027)
```

In [19]:

```
# class distribution (classes are imblanced )
fig = px.pie(Lymphoma, names='class', title='Lymphoma Class Distribution')
fig.update_traces(textposition='inside', textinfo='percent+label')
# Adjust the size of the pie chart
fig.update_layout(
    autosize=False,
    width=1000,
    height=600
)
fig.show()
```



Mapping classes :

- "DLBCL": 0 , "FL": 1 , "CLL": 2

Code:

```
: 1 # mapping classes
  2 Lymphoma_class=Lymphoma["class"]
  3 Lymphoma["class"] = Lymphoma["class"].map({"DLBCL":0,"FL":1,"CLL":2})
```

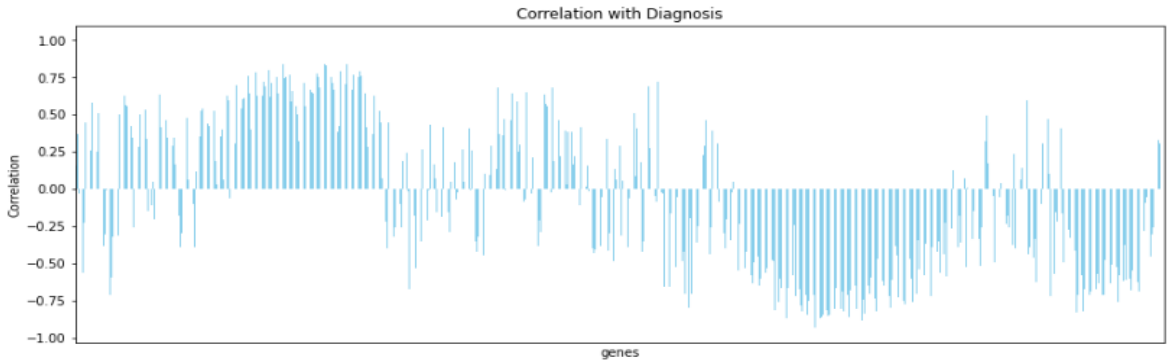
```
: 1 Lymphoma
```

:

ENE2534X	GENE2535X	GENE2536X	...	GENE3416X	GENE3829X	GENE3979X	GENE1489X	GENE1812X	GENE985X	GENE1809X	GENE1277X	GENE48X	class
-0.030000	-0.900000	0.510000	...	-0.290000	-0.590000	0.550000	-0.910000	1.080000	0.090000	0.000000	-0.020000	-0.040000	0
0.420000	0.240000	1.750000	...	0.130000	-0.620000	-0.040000	0.710000	0.710000	-0.310000	0.640000	0.540000	-0.140000	0
1.000000	1.670000	0.720000	...	0.680000	0.370000	-0.640000	3.250000	0.230000	0.210000	-0.330000	-0.620000	0.290000	0
-0.170000	-1.280000	0.340000	...	0.230000	-0.090000	0.610000	0.120000	0.430000	0.000000	0.140000	0.320000	0.050000	0
-0.280000	-0.400000	0.400000	...	0.050000	0.170000	0.210000	-0.610000	-0.920000	0.010000	0.500000	0.450000	-0.040000	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
-0.030218	-0.369390	-0.511930	...	-0.256909	0.390493	-0.268897	-0.636357	-1.252598	-0.118897	-0.898897	-0.421713	-0.205748	1
0.216105	-0.148070	-0.474222	...	0.143622	0.074921	0.032865	0.042086	-0.424379	-0.255566	-0.244434	-0.166432	0.127244	1
0.305922	0.072778	-1.334583	...	-0.100669	0.664950	-0.242576	-0.186521	-0.339477	-0.301105	-1.124249	-0.020133	-0.205385	1
0.286944	-0.622675	0.012731	...	-0.300854	0.195674	-0.122505	-0.645977	-1.001045	-0.246888	-0.295201	-0.118179	-0.148539	1
0.091545	-0.579132	0.333168	...	-0.129594	0.310000	-0.748129	-0.237508	-1.219865	-0.718399	-1.584307	-0.222413	-0.291545	1

Correlation between features and target

In [38]:
```
corr = Lymphoma.corr()
class_corr = corr['class']
correlation=class_corr.sort_values(ascending = False)
plt.figure(figsize=(15, 5))
class_corr.plot(kind='bar', color='skyblue')
plt.title('Correlation with Diagnosis')
plt.xlabel('genes')
plt.ylabel('Correlation')
plt.xticks([])
plt.show()
```



In [39]:
```
# drop in columns which is high correlated with target
for column in Lymphoma.columns:
    correlation = Lymphoma[column].corr(Lymphoma['class'])  # Calculate the correlation
    if abs(correlation) < 0.5 and abs(correlation) > -0.5 :
        Lymphoma.drop(column, axis=1, inplace=True)
```

In [40]:
```
print('number of columns after deleting columns which is low correlated with target :',Lymphoma.shape[1])
```

```
number of columns after deleting columns which is low correlated with target : 1613
```

Classification

1. The classification_model function takes four arguments: model, which represents the classification model to be trained, data, which is a pandas DataFrame containing the training data, predictors, which is a list of column names in the data DataFrame that will be used as input features for the model, and outcome, which is the column name representing the target variable or the class labels.

2. Inside the classification_model function, the model is trained using the fit method. It takes the predictors (data[predictors]) and the outcome (data[outcome]) as inputs.

3. The trained model is then used to make predictions on the same set of predictors using the predict method. The predictions are stored in the predictions variable.

4. The accuracy of the model is calculated by comparing the predictions with the actual outcome values using the accuracy_score function. The resulting accuracy is then printed.

5. Next, the code performs k-fold cross-validation to assess the model's performance. The KFold function from the scikit-learn library is used to generate the indices for the train-test splits. The n_splits parameter is set to 5, indicating that the data will be split into 5 folds.

6. Within a for loop, the data is split into train and test sets using the indices obtained from KFold.split(). The model is trained on the training data and then evaluated on the test data. The model's score, which represents the accuracy, is appended to the error list.

7. After the loop, the average cross-validation score is calculated by taking the mean of the error list. This score is then printed.

8. Finally, the function fits the model again on the entire training data before returning.

9. The rest of the code outside the function defines the predictor variables (predictor_var) and the outcome variable (outcome_var) using a dataset.

10. An MLPClassifier model is created and assigned to the variable model.

11. The data is split into training and testing sets using the train_test_split function from scikit-learn. The test_size parameter is set to 0.3, indicating that 30% of the data will be used for testing. The random_state parameter is set to 42 for reproducibility.

12. The training data is created by concatenating the predictor variables (X_train) and the outcome variable (y_train) using pd.concat().

13. The classification_model function is called with the model, train_data, list of predictor columns, and the name of the outcome variable as arguments.

Code:

[16]:

```
def classification_model(model, data, predictors, outcome):

  model.fit(data[predictors], data[outcome])
  predictions = model.predict(data[predictors])
  accuracy = accuracy_score(predictions, data[outcome])
  print("Accuracy: %s" % "{0:.3%}".format(accuracy))

  kf = KFold(n_splits=5)
  error = []

  for train, test in kf.split(data):
    train_predictors = data[predictors].iloc[train, :]
    train_target = data[outcome].iloc[train]
    model.fit(train_predictors, train_target)
    error.append(model.score(data[predictors].iloc[test, :], data[outcome].iloc[test]))

  print("Cross-Validation Score: %s" % "{0:.3%}".format(np.mean(error)))

  model.fit(data[predictors], data[outcome])

predictor_var = Lymphoma.drop('class', axis=1)
outcome_var = Lymphoma['class']

model = MLPClassifier()

# Shuffle and split the data
X_train, X_test, y_train, y_test = train_test_split(predictor_var, outcome_var, test_size=0.3, random_state=42, shuffle=True

train_data = pd.concat([X_train, y_train], axis=1)
classification_model(model, train_data, list(predictor_var.columns), 'class')
```

```
Accuracy: 100.000%
Cross-Validation Score: 98.947%
```

k-nearest neighbors (KNN) algorithm:

This code performs hyperparameter tuning for a k-nearest neighbors (KNN) classifier using the GridSearchCV class from scikit-learn. The goal is to find the best set of hyperparameters that maximize the accuracy of the model.

1. The param_grid dictionary defines the parameter grid for the GridSearchCV. In this case, it specifies different values for the n_neighbors hyperparameter, which represents the number of neighbors to consider in the KNN algorithm.
2. The GridSearchCV object is instantiated with the following parameters:
   - estimator: The base classifier (knn_classifier).
   - param_grid: The parameter grid to search over (param_grid).
   - cv: The number of folds for cross-validation. In this case, it is set to 5, meaning that the data will be split into 5 folds.
   - scoring: The scoring metric used to evaluate the performance of the model. Here, it is set to 'accuracy' to optimize for classification accuracy.
3. The fit method is called on the grid search object, passing in the training data (X_train and y_train). This fits the grid search to the data, which means it will perform an exhaustive search over the specified parameter grid, evaluating the model's performance using cross-validation.
4. After the grid search is fitted to the data, the best parameters and the corresponding best score are obtained. The best_params_ attribute of the grid search object stores the best parameters found during the search, and the best_score_ attribute stores the corresponding best score and finally, the best parameters and the best accuracy score are printed.

Code:

n [17]:

```
# Define the parameter grid for GridSearchCV
param_grid = {
    'n_neighbors': [3, 5, 7, 9],
}

# Instantiate the KNN classifier
knn_classifier = KNeighborsClassifier()

# Create GridSearchCV object
grid_search = GridSearchCV(knn_classifier, param_grid, cv=5, scoring='accuracy')

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Get the best parameters and the best score
best_params = grid_search.best_params_
best_score = grid_search.best_score_

print("Best Parameters:", best_params)
print("Best Accuracy:", best_score)
```

```
Best Parameters: {'n_neighbors': 3}
Best Accuracy: 1.0
```

After that the best parameters obtained from the GridSearchCV are used to instantiate a new KNN classifier. The classifier is then fitted on the training data, and its accuracy is evaluated on both the training and test sets.

1. The best parameters obtained from the GridSearchCV, {'n_neighbors': 3}, are stored in the best_params dictionary.
2. A new instance of the KNeighborsClassifier class is created and assigned to the variable knn_classifier. The **best_params syntax is used to pass the best parameters as keyword arguments to the classifier. In this case, the n_neighbors hyperparameter is set to 3.
3. The fit method is called on the knn_classifier object, passing in the training data (X_train and y_train). This trains the KNN classifier on the training data using the best parameters.
4. The score method is called on the knn_classifier object to calculate the accuracy of the model on the training data. The resulting accuracy is stored in the train_accuracy variable.
5. The score method is again called on the knn_classifier object, this time passing in the test data (X_test and y_test). This calculates the accuracy of the model on the test data, and the resulting accuracy is stored in the test_accuracy variable.
6. Finally, the training accuracy and test accuracy with the best parameters are printed.

Code:

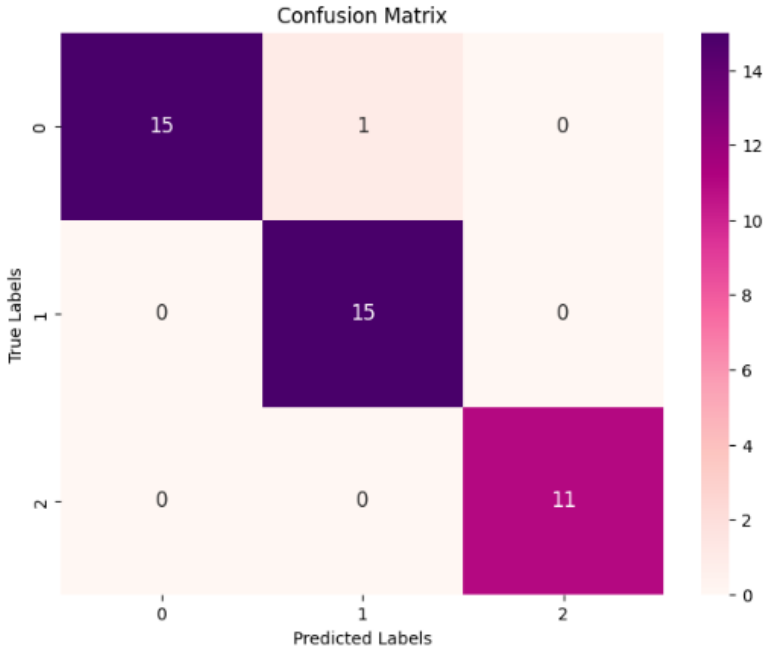
In [18]:

```
# Best parameters obtained from GridSearchCV
best_params = {'n_neighbors': 3}
# Instantiate the KNN classifier with the best parameters
knn_classifier = KNeighborsClassifier(**best_params)
# Fit the classifier on the training data
knn_classifier.fit(X_train, y_train)
# Evaluate the model on training and test sets
train_accuracy = knn_classifier.score(X_train, y_train)
test_accuracy = knn_classifier.score(X_test, y_test)
print("Training Accuracy with Best Parameters:", train_accuracy)
print("Test Accuracy with Best Parameters:", test_accuracy)
```

```
Training Accuracy with Best Parameters: 1.0
Test Accuracy with Best Parameters: 0.9761904761904762
```

Confusion matrix :

In [19]:
```
# Predict using the fitted classifier on the test set
knn_pred = knn_classifier.predict(X_test)

# Calculate the confusion matrix
knn_cm = confusion_matrix(y_test, knn_pred)

# Visualize the confusion matrix as a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(knn_cm, annot=True, fmt='d', cmap='RdPu', annot_kws={"size": 12})
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```



KNeighborsClassifier report code :

n [20]:
```
print('\nKNeighborsClassifier report : \n\n',classification_report(y_test,knn_pred))
accuracy = accuracy_score(y_test, knn_pred)

print("Accuracy Score:", accuracy)

```

```
KNeighborsClassifier report :

              precision    recall  f1-score   support

           0       1.00      0.94      0.97        16
           1       0.94      1.00      0.97        15
           2       1.00      1.00      1.00        11

    accuracy                           0.98        42
   macro avg       0.98      0.98      0.98        42
weighted avg       0.98      0.98      0.98        42

Accuracy Score: 0.9761904761904762
```

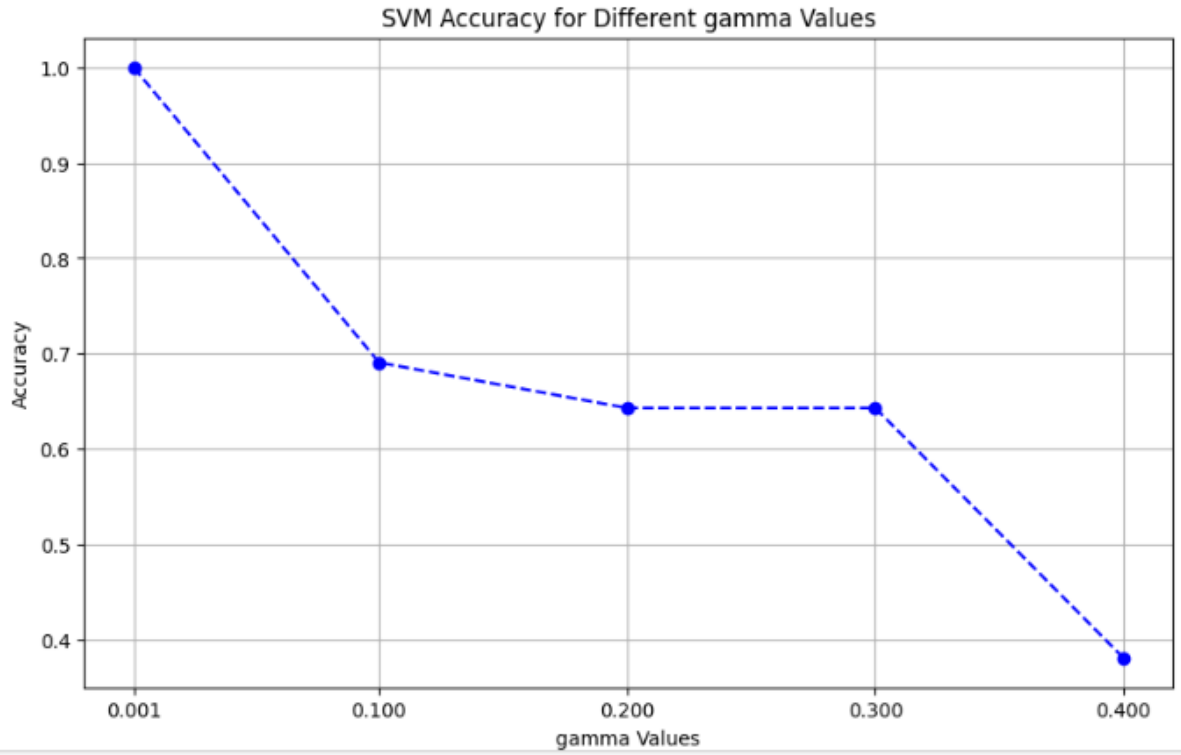
Support Vector Machine (SVM) algorithm:

The code provided different values of the gamma parameter are used to train Support Vector Machine (SVM) classifiers, and the corresponding accuracy scores are calculated and stored. The accuracy scores are then visualized using a line plot.

Code:

In [21]:

```
# Different C values to visualize
gamma_values = [0.001, 0.1, 0.2, 0.3,0.4]
accuracy = []
for gamma in gamma_values:
    svm_classifier = SVC(gamma=gamma)
    svm_classifier.fit(X_train, y_train)
    accuracy.append(svm_classifier.score(X_test, y_test))

# Visualize the accuracy for different C values
plt.figure(figsize=(10, 6))
plt.plot(gamma_values, accuracy, marker='o', linestyle='dashed', color='b')
plt.title('SVM Accuracy for Different gamma Values')
plt.xlabel('gamma Values')
plt.ylabel('Accuracy')
plt.xticks(gamma_values)
plt.grid(True)
plt.show()
```

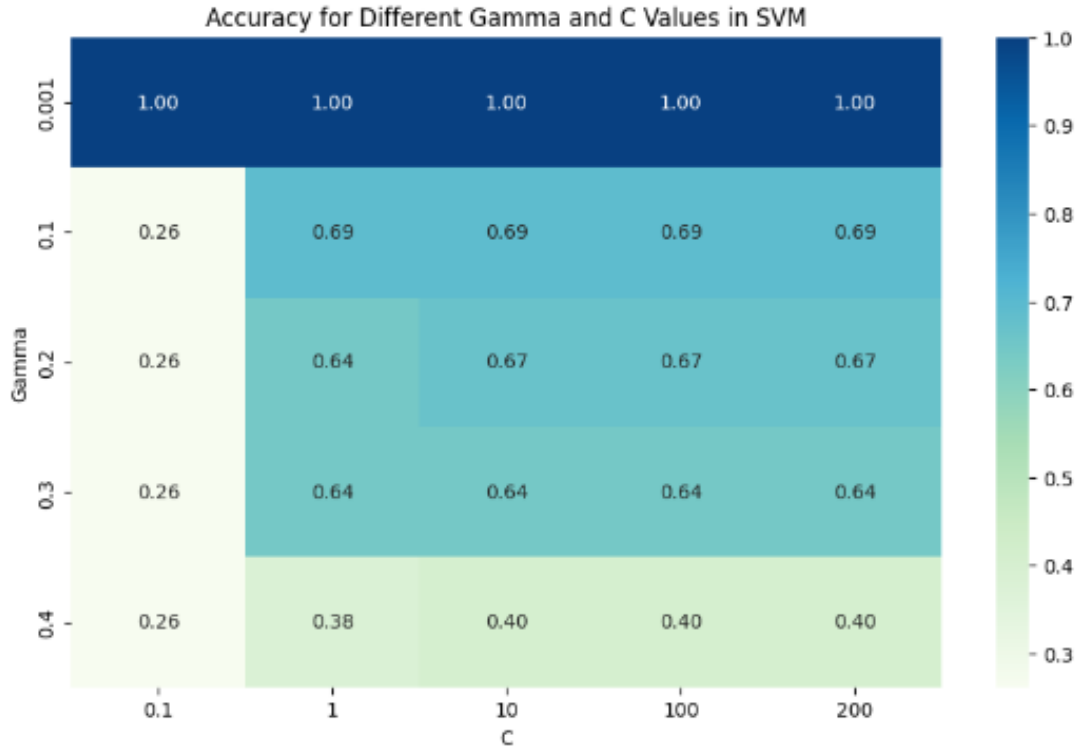


And this code also provided different values of both the gamma and C parameters are tested for Support Vector Machine (SVM) classifiers. But the accuracy scores for each combination of gamma and C values are calculated and stored in a matrix. The matrix is then visualized using a heatmap.

Code:

In [22]:

```
# Different gamma and C values to test
gamma_values = [0.001, 0.1, 0.2, 0.3,0.4]
C_values = [0.1, 1, 10, 100, 200]

accuracy_matrix = []

for gamma in gamma_values:
    accuracy = []
    for C in C_values:
        svm_classifier = SVC(gamma=gamma, C=C)
        svm_classifier.fit(X_train, y_train)
        svm_accuracy = svm_classifier.score(X_test, y_test)
        accuracy.append(svm_accuracy)
    accuracy_matrix.append(accuracy)

accuracy_matrix = np.array(accuracy_matrix)

plt.figure(figsize=(10, 6))
sns.heatmap(accuracy_matrix, annot=True, fmt='.2f', cmap='GnBu',
            xticklabels=C_values, yticklabels=gamma_values)
plt.title('Accuracy for Different Gamma and C Values in SVM')
plt.xlabel('C')
plt.ylabel('Gamma')
plt.show()
```



A Support Vector Machine (SVM) classifier is initialized and trained with specified parameters (gamma=0.001 and C=1). The accuracy of the trained classifier is then calculated and printed for both the training and test sets.

1. An instance of the SVC (Support Vector Classifier) class is created and assigned to the variable svm_classifier. The gamma parameter is set to 0.001, and the C parameter is set to 1.
2. The svm_classifier is trained using the training data (X_train and y_train) by calling the fit method.
3. The accuracy of the trained svm_classifier is calculated on the training set using the score method, which takes the training data as input. The resulting accuracy is stored in the train_accuracy variable.
4. The accuracy of the trained svm_classifier is calculated on the test set using the score method, which takes the test data (X_test and y_test) as input. The resulting accuracy is stored in the test_accuracy variable.

Code:

3]:

```
# Initialize and train the SVM classifier with specified parameters
svm_classifier = SVC(gamma=0.001, C=1)
svm_classifier.fit(X_train, y_train)

# Calculate accuracy on training set
train_accuracy = svm_classifier.score(X_train, y_train)

# Calculate accuracy on test set
test_accuracy = svm_classifier.score(X_test, y_test)

print("Support Vector Machine (SVM) - Training Accuracy:", train_accuracy)
print("Support Vector Machine (SVM) - Test Accuracy:", test_accuracy)
```

```
Support Vector Machine (SVM) - Training Accuracy: 1.0
Support Vector Machine (SVM) - Test Accuracy: 1.0
```

SVM_classifier report code:

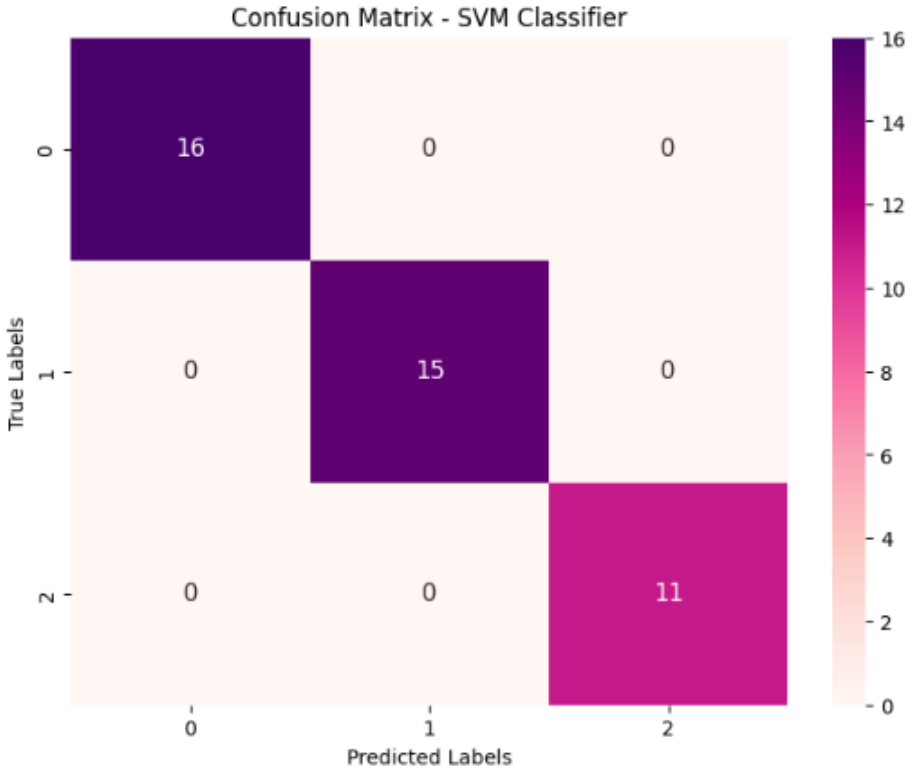
In [26]:

```
print('\nSVMClassifier report : \n\n',classification_report(y_test,svm_pred))
accuracy = accuracy_score(y_test, svm_pred)
print("Accuracy Score:", accuracy)
```

```
SVMClassifier report :

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        16
           1       1.00      1.00      1.00        15
           2       1.00      1.00      1.00        11

    accuracy                           1.00        42
   macro avg       1.00      1.00      1.00        42
weighted avg       1.00      1.00      1.00        42

Accuracy Score: 1.0
```

Confusion matrix : predictions are made on the test set using the trained Support Vector Machine (SVM) classifier. A confusion matrix is then generated based on the predicted labels and true labels, and it is displayed using a heatmap.

In [24]:
```
# Make predictions on the test set
svm_pred = svm_classifier.predict(X_test)

# Generate confusion matrix
svm_cm = confusion_matrix(y_test, svm_pred)

# Display the confusion matrix using a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(svm_cm, annot=True, fmt='d', cmap='RdPu', annot_kws={"size": 12})
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix - SVM Classifier')
plt.show()
```



Neural network algorithm:

This code performs hyperparameter tuning for a neural network classifier using grid search with cross-validation.

1. Parameter Grid Definition: The param_grid variable is a dictionary that defines the parameter grid to search. It specifies the different values to explore for two hyperparameters: 'activation' and 'alpha'.
   - For the 'activation' hyperparameter, the code specifies four activation functions: 'relu', 'tanh', 'logistic', and 'identity'.
   - For the 'alpha' hyperparameter, the code specifies five different values: 0.0001, 0.001, 0.01, 0.1, and 0.2.
2. MLPClassifier Creation:The code creates an instance of the MLPClassifier class, which represents a neural network classifier.
   - The random_state parameter is set to 42, which ensures reproducibility of results.
   - The max_iter parameter is set to 100, which determines the maximum number of iterations (epochs) the neural network will train for.
3. Grid Search with Cross-Validation:The code creates a GridSearchCV object named grid_search.
   - The GridSearchCV class performs an exhaustive search over the specified parameter grid using cross-validation.
   - It takes three arguments: the classifier (nn_classifier), the parameter grid (param_grid), and the number of cross-validation folds (cv=5).
   - The X_train and y_train variables are assumed to contain the training data and labels, respectively.
   - The fit() method is called on the grid_search object to perform the grid search and cross-validation on the training data.
4. Results Extraction:After the grid search is completed, the code extracts the results from the grid_search object.
   - The cv_results_ attribute of the grid_search object contains a dictionary with information about the search results.
   - The results variable is assigned the value of grid_search.cv_results_.

5. Mean Test Scores Extraction: The code extracts the mean test scores from the results dictionary.
   - The 'mean_test_score' key in the results dictionary contains an array of mean test scores for each combination of hyperparameters.
   - The np.array() function is used to convert the mean test scores into a NumPy array.
   - The reshape() method is applied to the array to reshape it into a 2D array with dimensions based on the number of activation functions and alpha values in the parameter grid.

Code:

In [26]:

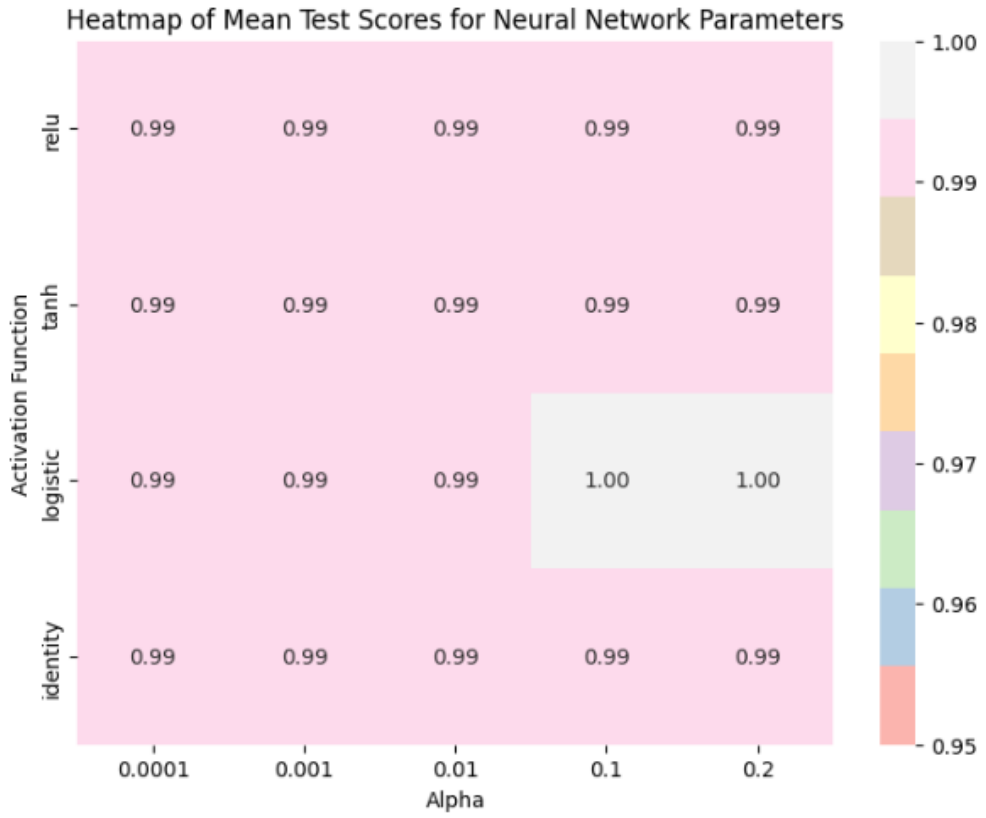
```
# Define the parameter grid to search
param_grid = {
    'activation': ['relu', 'tanh', 'logistic', 'identity'],
    'alpha': [0.0001, 0.001, 0.01, 0.1, 0.2],
}

# Create an MLPClassifier
nn_classifier = MLPClassifier(random_state=42, max_iter=100)

# Grid search with cross-validation
grid_search = GridSearchCV(nn_classifier, param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Extract results from grid search
results = grid_search.cv_results_

# Extract mean test scores from results
mean_test_scores = np.array(results['mean_test_score']).reshape(len(param_grid['activation']), -1)
```

This code snippet uses the Seaborn library to create a heatmap based on the mean_test_scores array.

Code:

n [27]:

```
# Create a heatmap using Seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(mean_test_scores, annot=True, fmt='.2f', cmap='Pastel1',
            xticklabels=param_grid['alpha'], yticklabels=param_grid['activation'], vmin=0.95, vmax=1.0)
plt.title('Heatmap of Mean Test Scores for Neural Network Parameters')
plt.xlabel('Alpha')
plt.ylabel('Activation Function')
plt.show()
```

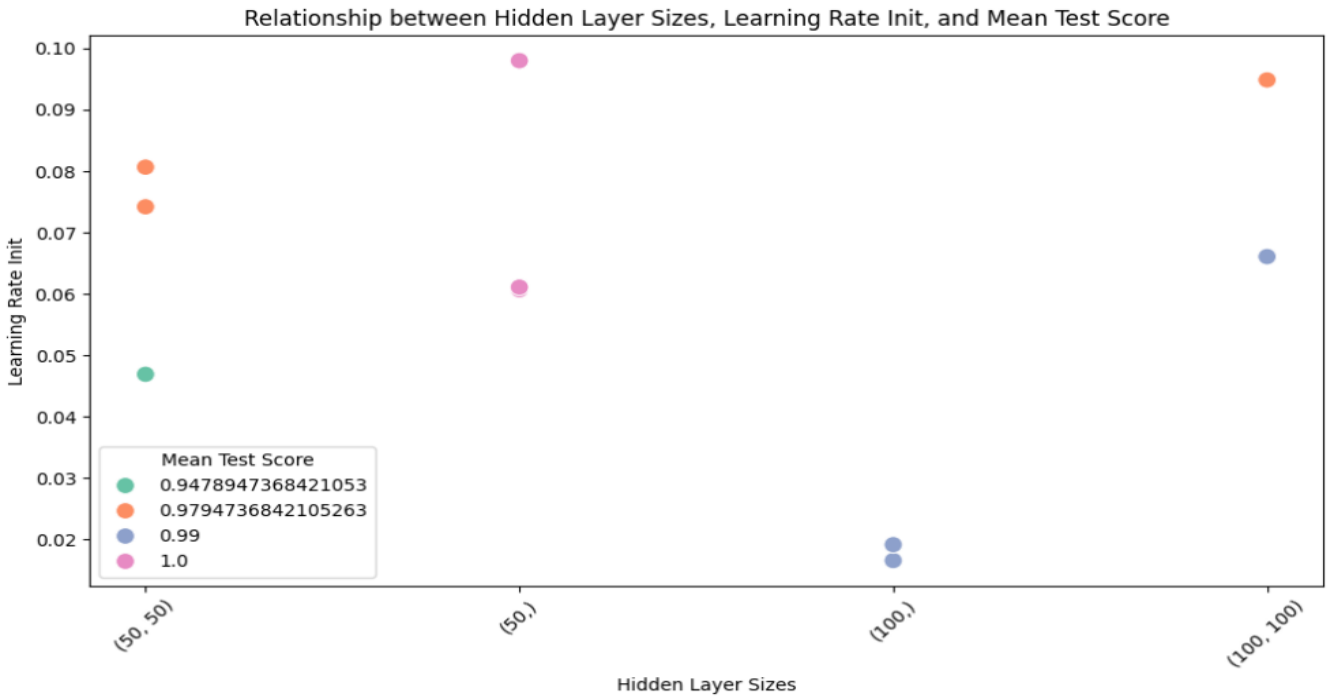


After that we use a code to snippet performs a randomized search with cross-validation to find the best hyperparameters for an MLPClassifier. It then creates a scatter plot to visualize the relationship between the selected hyperparameters and the mean test scores.

Code:

[28]:

```
# Define the parameter distributions for randomized search
param_dist = {
    'hidden_layer_sizes': [(50,), (100,), (50, 50), (100, 100)],  # Example hidden layer sizes
    'learning_rate_init': uniform(0.001, 0.1),  # Range of learning rate values
}

# Create an MLPClassifier
nn_classifier = MLPClassifier(random_state=42, max_iter=100)

# Randomized search with cross-validation
random_search = RandomizedSearchCV(nn_classifier, param_distributions=param_dist, n_iter=10, cv=5, random_state=42)
random_search.fit(X_train, y_train)

# Extract best parameters from randomized search
best_params = random_search.best_params_

# Extract hidden layer sizes and learning rate init values used in the search
hidden_layer_sizes = random_search.cv_results_['param_hidden_layer_sizes'].data
learning_rate_init_values = random_search.cv_results_['param_learning_rate_init'].data

# Create a DataFrame for scatter plot data
results_df = pd.DataFrame({
    'Hidden Layer Sizes': [str(layer_size) for layer_size in hidden_layer_sizes],
    'Learning Rate Init': learning_rate_init_values,
    'Mean Test Score': random_search.cv_results_['mean_test_score']
})

# Create a scatter plot
plt.figure(figsize=(10, 6))
sns.scatterplot(data=results_df, x='Hidden Layer Sizes', y='Learning Rate Init', hue='Mean Test Score', palette='Set2', s=10
plt.title('Relationship between Hidden Layer Sizes, Learning Rate Init, and Mean Test Score')
plt.xlabel('Hidden Layer Sizes')
plt.ylabel('Learning Rate Init')
plt.legend(title='Mean Test Score')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Best parameters from the randomized search
print("Best Parameters:", best_params)
```



This code snippet defines specific hyperparameters for an MLPClassifier, creates and trains the classifier using the defined hyperparameters, and calculates the accuracy on both the training and test sets. Here's a breakdown of what each section does:

1. Hyperparameter Definition: The params dictionary contains specific hyperparameter values for the MLPClassifier.
   - The 'hidden_layer_sizes' key specifies a tuple (50,) as the size of the hidden layer.
   - The 'alpha' key sets the regularization parameter to 0.1.
   - The 'learning_rate_init' key sets the initial learning rate to 0.0606850157946487.
   - The 'random_state' key sets the random seed to 42.
   - The 'activation' key sets the activation function to 'logistic'.
2. MLPClassifier Creation and Training: An instance of the MLPClassifier class is created, passing the hyperparameters defined in the params dictionary using the ** syntax to unpack the dictionary.
   - The fit() method is called on the nn_classifier object to train the classifier using the training data X_train and y_train.
3. Accuracy Calculation: The score() method is called on the nn_classifier object with X_train and y_train as arguments to calculate the accuracy on the training set. The result is stored in the train_accuracy variable.
   - The score() method is called again on the nn_classifier object with X_test and y_test as arguments to calculate the accuracy on the test set. The result is stored in the test_accuracy variable.

Code:

In [29]:

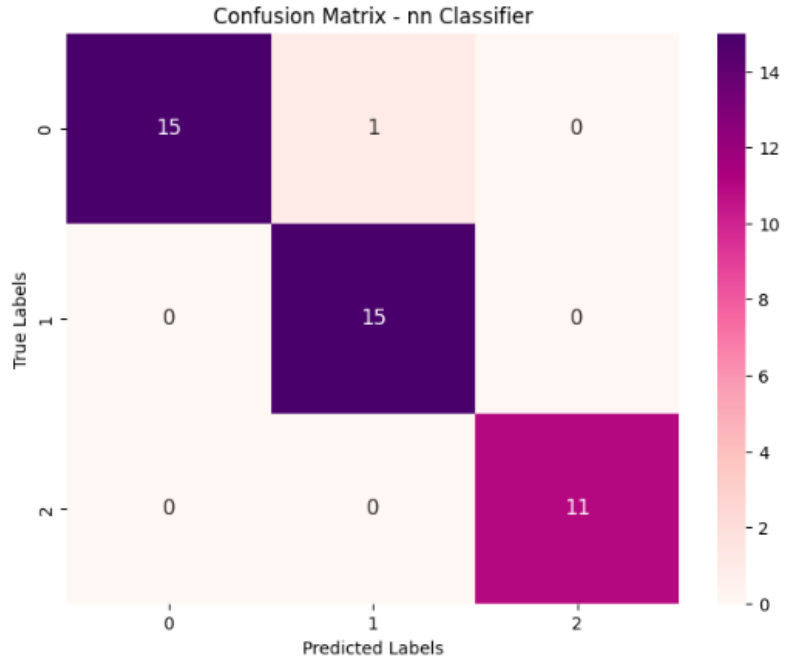
```
# Define specific hyperparameters
params = {
    'hidden_layer_sizes': (50,),
    'alpha': 0.1,
    'learning_rate_init': 0.0606850157946487,
    'random_state': 42,
    'activation': 'logistic'
}

# Create and train the MLPClassifier
nn_classifier = MLPClassifier(**params)
nn_classifier.fit(X_train, y_train)
# Calculate accuracy on training set
train_accuracy = nn_classifier.score(X_train, y_train)

# Calculate accuracy on test set
test_accuracy = nn_classifier.score(X_test, y_test)

print("Multi-Layer Perceptron (MLP) - Training Accuracy: ", train_accuracy)
print("Multi-Layer Perceptron (MLP) - Test Accuracy: ", test_accuracy)
```

```
Multi-Layer Perceptron (MLP) - Training Accuracy:  1.0
Multi-Layer Perceptron (MLP) - Test Accuracy:  0.9761904761904762
```

Confusion matrix : makes predictions on the test set using the trained MLPClassifier, generates a confusion matrix, and displays the confusion matrix using a heatmap.

Code for Confusion matrix and report:

In [30]:
```
# Make predictions on the test set
nn_pred = nn_classifier.predict(X_test)

# Generate confusion matrix
nn_cm = confusion_matrix(y_test, nn_pred)

# Display the confusion matrix using a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(nn_cm, annot=True, fmt='d', cmap='RdPu', annot_kws={"size": 12})
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix - nn Classifier')
plt.show()
```



n [20]:
```
print('\nKNeighborsClassifier report : \n\n',classification_report(y_test,knn_pred))
accuracy = accuracy_score(y_test, knn_pred)

print("Accuracy Score:", accuracy)

```

```
KNeighborsClassifier report :

              precision    recall  f1-score   support

           0       1.00      0.94      0.97        16
           1       0.94      1.00      0.97        15
           2       1.00      1.00      1.00        11

    accuracy                           0.98        42
   macro avg       0.98      0.98      0.98        42
weighted avg       0.98      0.98      0.98        42

Accuracy Score: 0.9761904761904762
```

Feature Selection Using Filter Methods:

Methods used in the project:

- Variance Threshold
- Correlation-based
- feature importance using Random Forest Classifier

1- Variance Threshold

VarianceThreshold is a feature selection technique provided by scikit-learn that removes features (columns) with low variance. In other words, it filters out features that have little variation in their values across the samples in the dataset. This is particularly useful when dealing with datasets where some features have nearly constant values, as these features may not contribute much to the modeling process.

Here's a brief overview of how **VarianceThreshold** works:

- **Calculate Variance:**
  - For each feature, **VarianceThreshold** calculates the variance across all samples. Variance is a measure of how much values in a feature vary from the mean.
- **Remove Features Below Threshold:**
  - Features with variance below a specified threshold are considered to have low variability and are removed from the dataset.
- **Retain Features Above Threshold:**
  - Features with variance equal to or above the threshold are retained in the dataset.

A- VarianceThreshold

```
X = Lymphoma.drop(columns=['class'])
y = Lymphoma['class']
model = RandomForestClassifier()
thresholds = [0.01, 0.02, 0.05, 0.1, 0.2]
best_threshold = None
best_score = 0

for threshold in thresholds:
    # Feature selection using variance threshold
    selector = VarianceThreshold(threshold=threshold)
    X_selected = selector.fit_transform(X)

    # Use a classifier of your choice (e.g., RandomForestClassifier)
    # Evaluate model performance using cross-validation
    scores = cross_val_score(model, X_selected, y, cv=5)

    # Track the best threshold
    if scores.mean() > best_score:
        best_score = scores.mean()
        best_threshold = threshold

print(f"Best Threshold: {best_threshold}, Best Accuracy: {best_score}")
```

```
Best Threshold: 0.01, Best Accuracy: 1.0
```

- Assuming **Lymphoma** is a DataFrame, extract the feature matrix **X** (all columns except the last one) and the target variable **y** (last column, presumably 'class').
- define the threshold value for the variance. Features with variance below this threshold will be removed.
- Create an instance of the **VarianceThreshold** class with the specified threshold
- Use the **fit_transform** method to apply feature selection based on variance. It removes features with variance below the threshold and returns the filtered data
- Retrieve the indices of the selected features. These indices correspond to the columns in the original feature matrix **X**.
- Extract the names of the selected genes by indexing the original feature matrix columns using the selected indices

[69]:
```
# VarianceThreshold
import pandas as pd
from sklearn.feature_selection import VarianceThreshold
X = Lymphoma.iloc[:, :-1]  # Features
y = Lymphoma['class']  # Target variable
threshold = 0.1
selector = VarianceThreshold(threshold=threshold)
filtered_gene_data = selector.fit_transform(X)
selected_indices = selector.get_support(indices=True)
selected_genes = X.columns[selected_indices]
print("Selected Genes:")
print(selected_genes)
```

```
Selected Genes:
Index(['GENE1835X', 'GENE1836X', 'GENE1865X', 'GENE1380X', 'GENE1933X',
       'GENE1932X', 'GENE1931X', 'GENE1930X', 'GENE3129X', 'GENE3126X',
       ...
       'GENE4022X', 'GENE3931X', 'GENE2588X', 'GENE3120X', 'GENE6X', 'GENE5X',
       'GENE3X', 'GENE2X', 'GENE48X', 'GENE47X'],
      dtype='object', length=3950)
```

Here test this feature selection method with KNN model.

```
# Method 1: VarianceThreshold
X_var = X[selected_genes]
X_var_train, X_var_test, y_train, y_test = train_test_split(X_var, y, test_size=0.3, random_state=42)

# Create a KNN model with VarianceThreshold features
model_var = KNeighborsClassifier(n_neighbors=5)
model_var.fit(X_var_train, y_train)
y_pred_var = model_var.predict(X_var_test)
accuracy_var = accuracy_score(y_test, y_pred_var) * 100
print(f'Accuracy with VarianceThreshold Features: {accuracy_var:.2f}%')
```

```
Accuracy with VarianceThreshold Features: 97.62%
```

2- Correlation-based

Correlation-based feature selection is a technique used to identify and retain a subset of features in a dataset that exhibit a certain level of correlation with the target variable or with other features. The primary goal is to reduce the dimensionality of the dataset while preserving relevant information.

- Calculate the correlation matrix (**cor_matrix**) for the features in the dataset using the **corr** method.

- Define a correlation threshold (e.g., 0.8). Features with absolute correlation greater than this threshold are considered highly correlated.
- Use NumPy to find indices where the absolute correlation values are greater than the threshold.
- Iterate through the pairs of highly correlated features and add one of each pair to the set of features to be removed. This step ensures that only one feature from each highly correlated pair is added to the set.
- Create a new DataFrame (**selected_features**) by dropping the columns identified for removal from the original feature matrix **X**.

```
# Correlation-based Feature Selection
import pandas as pd
import numpy as np
X = Lymphoma.iloc[:, :-1]  # Features
y = Lymphoma['class']  # Target variable
cor_matrix = X.corr()
correlation_threshold = 0.8
highly_correlated_features = np.where(np.abs(cor_matrix) > correlation_threshold)
features_to_remove = set()
for i, j in zip(*highly_correlated_features):
    if i != j and i not in features_to_remove and j not in features_to_remove:
        features_to_remove.add(j)
selected_features = X.drop(columns=X.columns[list(features_to_remove)])

print("Selected Features:")
print(selected_features.head())
```

```
Selected Features:
   GENE3385X  GENE1114X  GENE3994X  GENE1140X  GENE2533X  GENE2535X  \
0      -0.16      -0.03       0.40      -0.25      -0.06      -0.90
1      -2.09      -0.55       0.59      -2.03       0.37       0.24
2      -0.83      -0.10       1.39       0.72      -0.06       1.67
3      -0.44       0.06       0.10      -0.52      -0.27      -1.28
4      -0.53      -0.77      -0.02      -0.17      -0.32      -0.40

   GENE2536X  GENE2540X  GENE3866X  GENE3442X  ...  GENE1346X  GENE3416X  \
0       0.51      -0.46       0.12       0.05  ...      -0.86      -0.29
1       1.75      -0.04      -2.32      -0.07  ...      -0.33       0.13
2       0.72      -0.06       0.85       0.92  ...      -0.06       0.68
3       0.34       0.32       0.02      -0.07  ...       0.59       0.23
4       0.40      -0.33       0.06       0.39  ...      -0.01       0.05

   GENE3829X  GENE3979X  GENE1489X  GENE1812X  GENE985X  GENE1809X  GENE1277X  \
0      -0.59       0.55      -0.91       1.08      0.09       0.00      -0.02
1      -0.62      -0.04       0.71       0.71     -0.31       0.64       0.54
2       0.37      -0.64       3.25       0.23      0.21      -0.33      -0.62
3      -0.09       0.61       0.12       0.43      0.00       0.14       0.32
4       0.17       0.21      -0.61      -0.92      0.01       0.50       0.45

   GENE48X
0    -0.04
1    -0.14
2     0.29
3     0.05
4    -0.04
```

Then test model:

```
# Method 2: Correlation-based Feature Selection
X_corr = selected_features
X_corr_train, X_corr_test , y_train, y_test  = train_test_split(X_corr,y, test_size=0.3, random_state=42)

# Create a KNN model with Correlation-based features
model_corr = KNeighborsClassifier(n_neighbors=5)
model_corr.fit(X_corr_train, y_train)
y_pred_corr = model_corr.predict(X_corr_test)
accuracy_corr = accuracy_score(y_test, y_pred_corr) * 100
print(f'Accuracy with Correlation-based Features: {accuracy_corr:.2f}%')
```

```
Accuracy with Correlation-based Features: 97.62%
```

feature importance using Random Forest Classifier

Feature importance in the context of a Random Forest Classifier is a technique to assess the contribution of each feature in the model's predictive performance. Random Forest is an ensemble learning algorithm that consists of a collection of decision trees. Each tree is trained on a random subset of the data, and the predictions are combined through a voting or averaging mechanism.

Random Forest provides a built-in feature importance measure based on the impurity reduction (or Gini impurity) achieved by each feature during the construction of the trees. The basic idea is to measure how much each feature contributes to the model's ability to make accurate predictions.

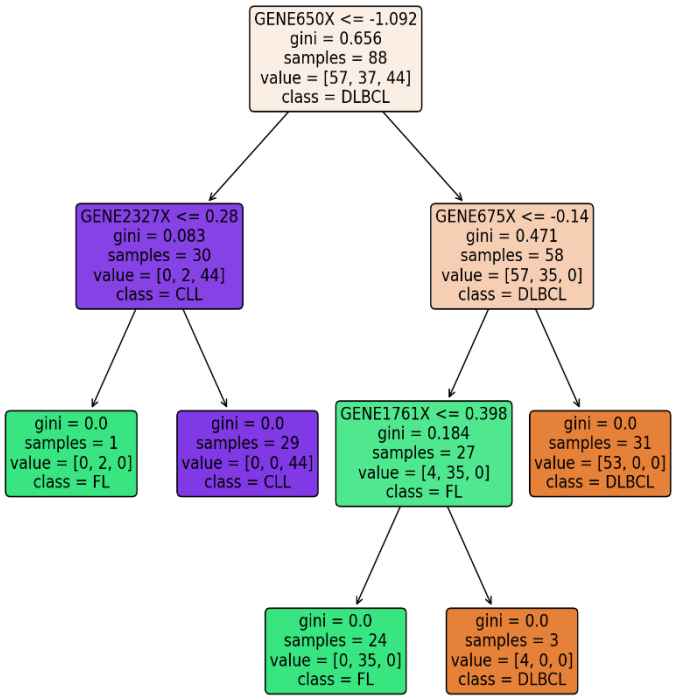
- Create an instance of the **RandomForestClassifier** class with 100 trees (**n_estimators**) and a fixed random seed (**random_state=42**).
- Train the Random Forest classifier on the feature matrix **X** and the target variable **y**.
- Retrieve the feature importances calculated by the trained Random Forest model.
- Create a DataFrame (**feature_importance_df**) to store the feature names and their corresponding importances.
- Sort the DataFrame by the importance scores in descending order to identify the most important features at the top.

```
]:
X =Lymphoma.drop('class', axis=1)  # Features
y =Lymphoma['class']  # Target variable
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
rf_classifier.fit(X, y)
feature_importances = rf_classifier.feature_importances_
feature_importance_df = pd.DataFrame({'Feature': X.columns, 'Importance': feature_importa
feature_importance_df = feature_importance_df.sort_values(by='Importance', ascending=Fals
print("Top Features and their Importance:")
print(feature_importance_df.head())
```

```
Top Features and their Importance:
        Feature  Importance
971    GENE653X    0.023488
256   GENE2403X    0.020808
260   GENE2399X    0.020298
180   GENE2395X    0.019923
1467  GENE1672X    0.019877
```

Visualize the tree:

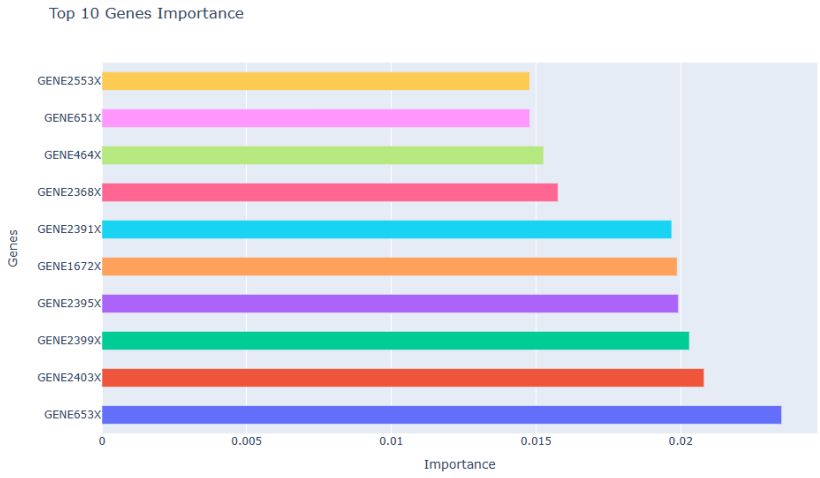
- **visualise_tree** is a function that takes a decision tree (**tree_to_print**) as input and visualizes it.
- Create a subplot within the figure with specified dimensions (10x10 inches) and resolution (dpi=200). The subplot axes are stored in the variables **fig** and **axes**.
- Use the **plot_tree** function from **sklearn.tree** to plot the decision tree (**tree_to_print**).
- **feature_names** is set to the names of the features in the dataset (**X.columns.tolist()**).
- **class_names** specifies the class labels as ['DLBCL', 'FL', 'CLL'].
- **filled=True** fills the decision tree nodes with color based on the majority class.
- **rounded=True** rounds the corners of the decision tree nodes in the visualization.




```
def visualise_tree(tree_to_print):
    plt.figure()
    fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (10,10), dpi=200)
    plot_tree(tree_to_print,
              feature_names = X.columns.tolist(),
              class_names=['DLBCL', 'FL', 'CLL'],
              filled = True,
             rounded=True);
    plt.show()
visualise_tree(rf_classifier.estimators_[3])
```

get the most important genes:

GENE653X is the most important gene.

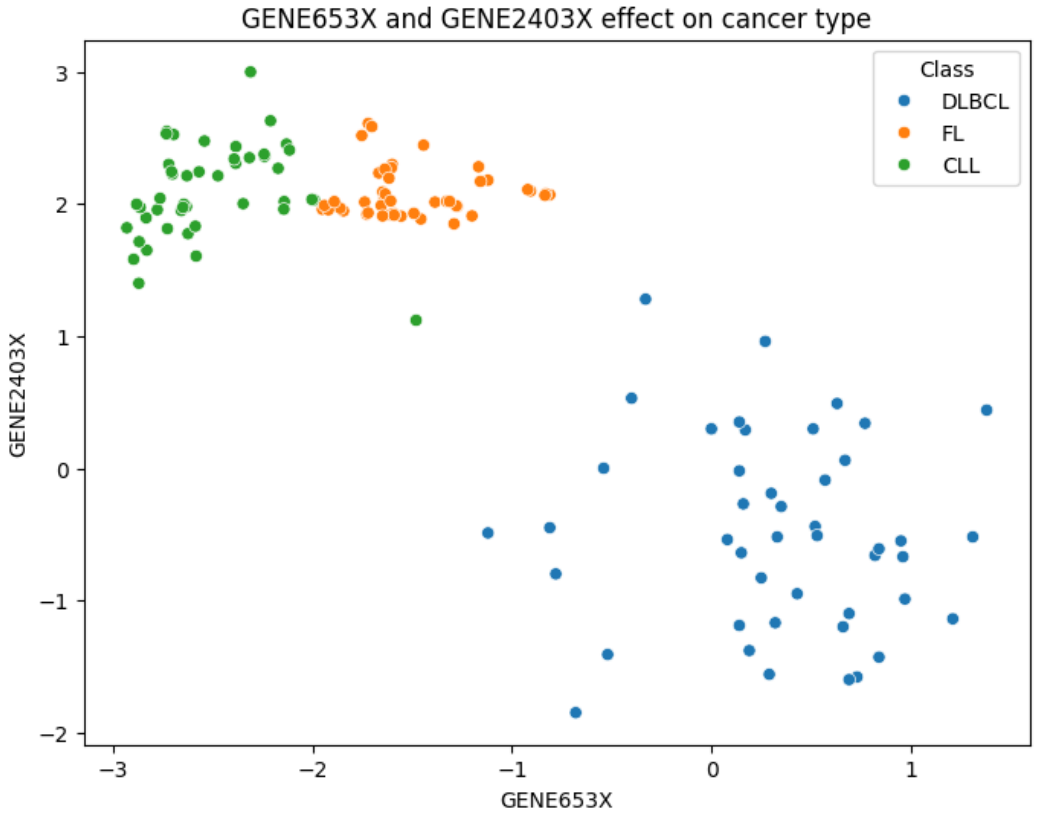


Selecting most important 2 genes to show it's relation with the target :

```
top_genes = feature_importance_df.head(2)['Feature'].tolist()
imp_df = X[top_genes].copy()

X_rf_train, X_rf_test, y_train, y_test = train_test_split(imp_df, y , test_size=0.3, random_state=42)

# Create a KNN model with RandomForest feature importance features
model_rf = KNeighborsClassifier(n_neighbors=5)
model_rf.fit(X_rf_train, y_train)
y_pred_rf = model_rf.predict(X_rf_test)
accuracy_rf = accuracy_score(y_test, y_pred_rf) * 100
print(f'Accuracy with Feature Importance Features: {accuracy_rf:.3f}%')
```

```
Accuracy with Feature Importance Features: 95.238%
```

```
plt.figure(figsize=(8, 6))
sns.scatterplot(x='GENE653X', y='GENE2403X', hue=Lymphoma_class, data=Lymphoma)
plt.title('GENE653X and GENE2403X effect on cancer type')
plt.xlabel('GENE653X')
plt.ylabel('GENE2403X')
plt.legend(title='Class')
plt.show()
```

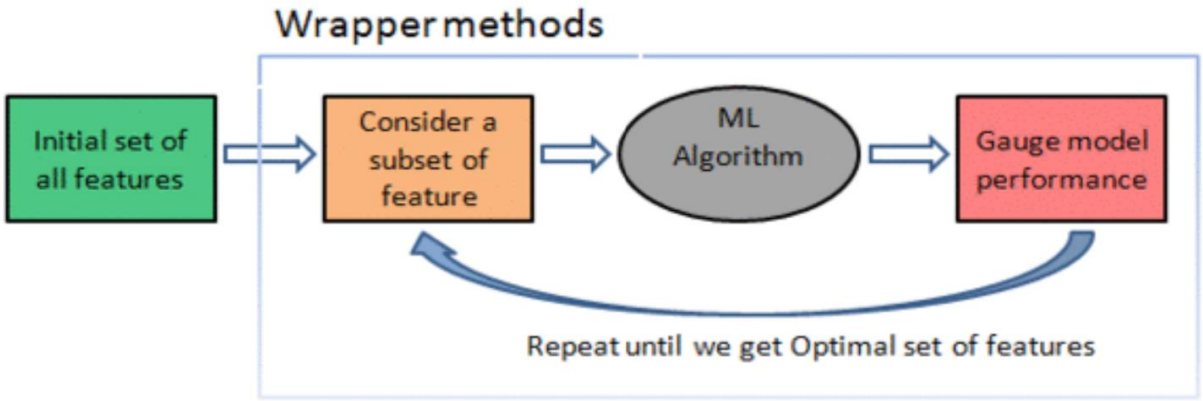


GENE653X effect the type of cancer more the second gene!

Feature Selection Using Wrapper methods:

In *wrapper methods,* the *feature selection* process is based on a specific machine learning algorithm that we are trying to fit on a given dataset.

In *wrapper methods,* the *feature selection* process is based on a specific machine learning algorithm that we are trying to fit on a given dataset.



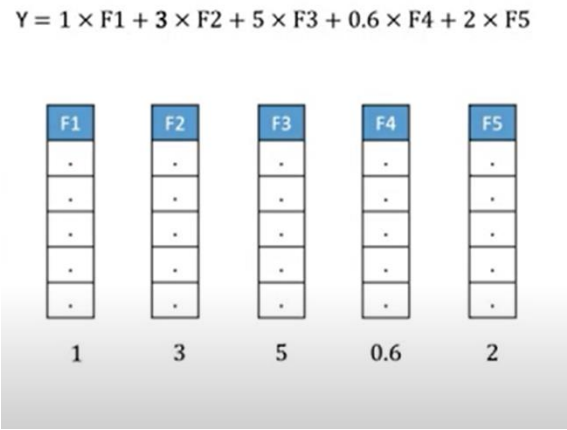
It follows a *greedy search approach* by evaluating all the possible combinations of features against the *evaluation criterion*. The *evaluation criterion* is simply the performance measure that depends on the type of problem, e.g. For *regression* evaluation criterion can be p-values, R-squared, or Adjusted R-squared, similarly for *classification* the evaluation criterion can be accuracy, precision, recall, f1-score, etc. Finally, it selects the combination of features that gives the optimal results for the specified machine learning algorithm.

Used Methods In project

1. Recursive feature elimination
2. Forward feature selection

1.Recursive Feature Elimination:-

Recursive feature elimination (RFE) is a feature selection method that fits a model and removes the weakest feature (or features) until the specified number of features is reached. Features are ranked by the model's coef_ or feature_importances_ attributes, and by recursively eliminating a small number of features per loop, RFE attempts to eliminate dependencies and collinearity that may exist in the model.



RFE requires a specified number of features to keep, however it is often not known in advance how many features are valid. To find the optimal number of features cross-validation is used with RFE to score different feature subsets and select the best scoring collection of features. The RFECV visualizer plots the number of features in the model along with their cross-validated test score and variability and visualizes the selected number of features.

```
model = RandomForestClassifier(n_estimators=100, random_state=42)

num_features_to_select = 30
rfe = RFE(estimator=model, n_features_to_select = num_features_to_select)

rfe.fit(X_train, y_train)

# ranking of each feature
feature_ranking = rfe.ranking_

selected_features = np.where(feature_ranking == 1)[0]

# Visualize the feature ranking
plt.figure(figsize=(10, 6))
plt.title("RFE - Feature Ranking")
plt.xlabel("Feature Index")
plt.ylabel("Ranking")
plt.bar(range(len(feature_ranking)), feature_ranking)
plt.show()

# Print the selected features
print("Selected Features:", selected_features)

# Train the final model using the selected features
model.fit(X_train.iloc[:, selected_features], y_train)

# Evaluate the model on the test set
accuracy = model.score(X_test.iloc[:, selected_features], y_test)
print("Accuracy on the Test Set:", accuracy)
```

1. ## Model initialization

   created a Random Forest Classifier (model) using sci-kit-learn's RandomForestClassifier. This classifier is an ensemble model that consists of multiple decision trees. The n_estimators parameter is set to 100, meaning your random forest will comprise 100 decision trees. The random_state parameter ensures reproducibility by fixing the random seed.

2. ## RFE initialization:

   initializing Recursive Feature Elimination (RFE) using scikit-learn's RFE class. RFE works by recursively fitting the model and eliminating the least important features until the desired number (num_features_to_select) is reached. we want to select 30 features.

3. ## Fit RFE on the training data:

   We applied RFE to the training data (X_train, y_train). This step involves fitting the model multiple times, ranking features based on their importance, and eliminating the least important ones.

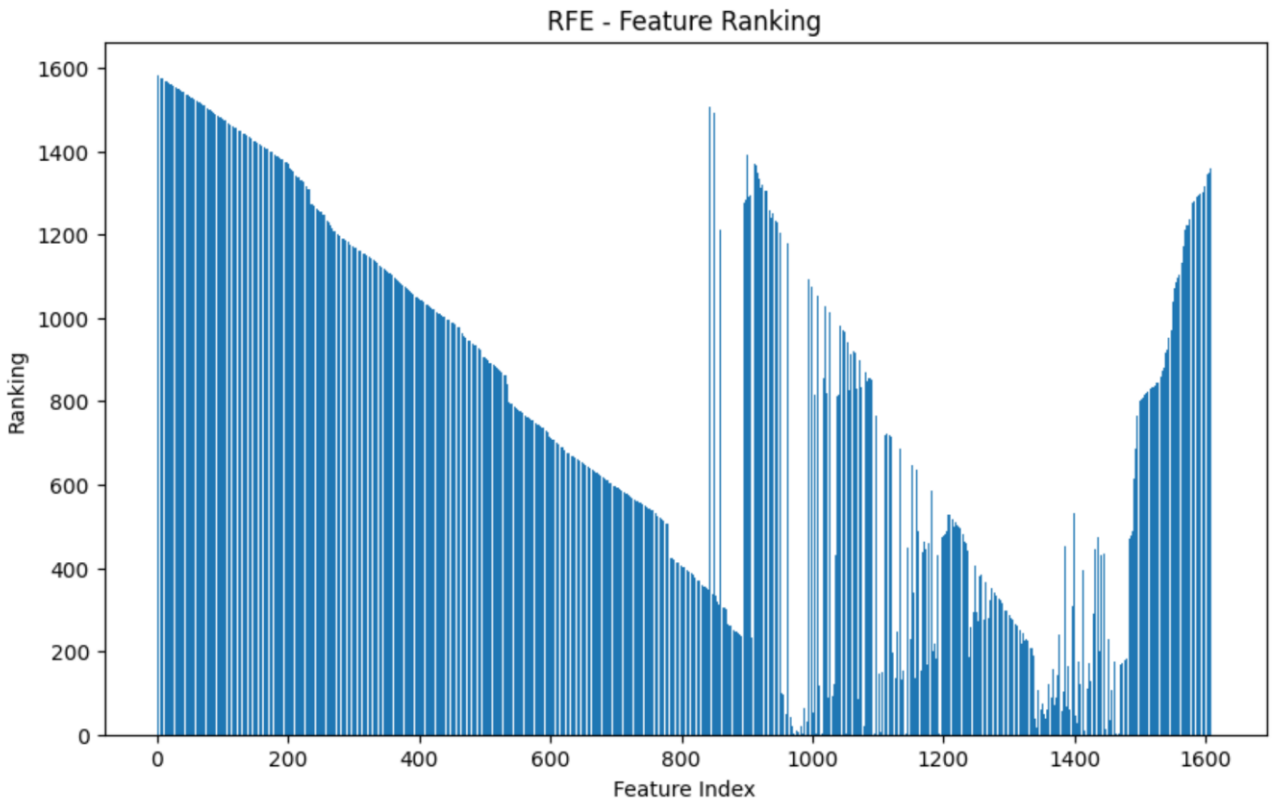
4. ## Get Feature Ranking:

   We obtained the ranking of each feature. Features with a ranking of 1 are considered the most important.

5. ## Identify the selected Features:-

   Then identifying the indices of the selected features by finding where the ranking is equal to 1.

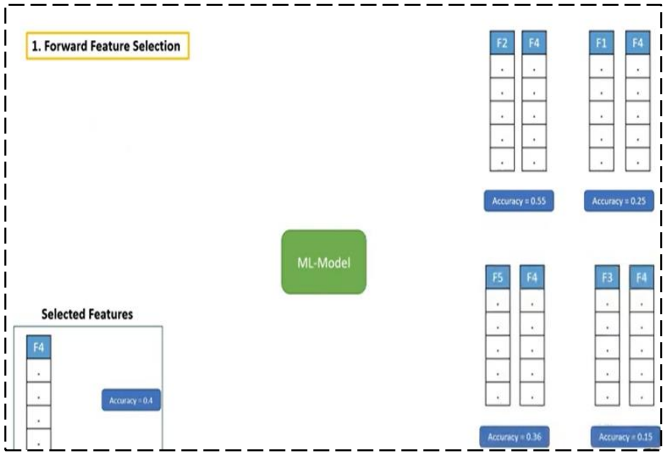
6. ## Visualize Feature Ranking:-



```
Selected Features: [ 971  972  973  974  977  985  986  989  990  993 1006 1011 1015 1032
 1051 1080 1092 1095 1101 1109 1143 1147 1156 1183 1193 1435 1459 1463
 1467 1468]
Accuracy on the Test Set: 1.0
```

2. Forward Feature Selection:-

In *forward selection*, we start with a null model and then start fitting the model with each individual feature one at a time and select the feature with the minimum *p-value*. Now fit a model with two features by trying combinations of the earlier selected feature with all other remaining features. Again select the feature with the minimum *p-value*. Now fit a model with three features by trying combinations of two previously selected features with other remaining features. Repeat this process until we have a set of selected features with a *p-value* of individual features less than the *significance level*.




```python
clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Sequential Forward Selection
sfs = SequentialFeatureSelector(clf, k_features='best', forward=True, scoring='accuracy', cv=5)
sfs.fit(X_train.iloc[:, selected_features], y_train)

# Selected features
selected_features = list(X_train.iloc[:, selected_features].columns[list(sfs.k_feature_idx_)])

# Train a model with the selected features
clf.fit(X_train[selected_features], y_train)

# Evaluate the model
accuracy = clf.score(X_test[selected_features], y_test)
print(f"Selected Features: {selected_features}")
print(f"Accuracy on Test Set: {accuracy}")
```

```
Selected Features: ['GENE669X', 'GENE598X']
Accuracy on Test Set: 0.9285714285714286
```

1. Model initialization:

initialize a Random Forest Classifier (clf) with 100 trees in the forest and a fixed random state for reproducibility.

2. SFS initialization: -

Sequential Forward Selection (SFS) is initialized using the scikit-learn library. The k_features='best' parameter indicates that the algorithm should select the best subset of features. forward=True specifies forward selection, where features are added one by one based on their contribution to the model. scoring='accuracy' sets the evaluation metric to accuracy, and cv=5 specifies 5-fold cross-validation during the selection process.

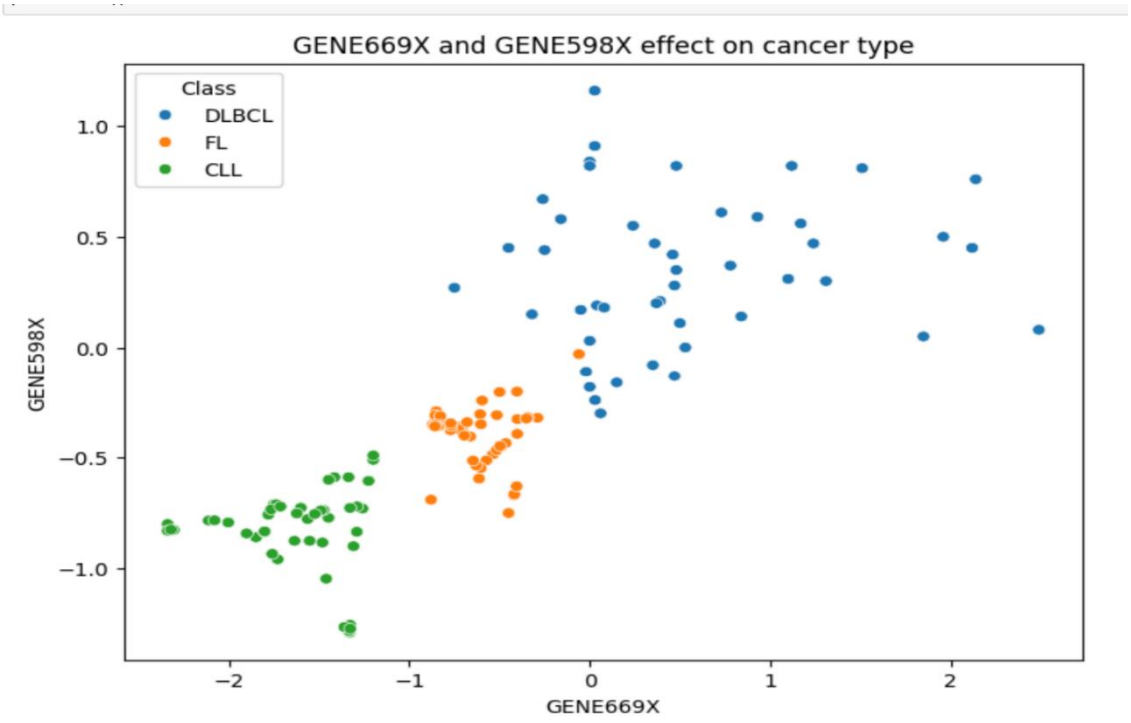
3. it is the SFS algorithm on the features selected by a previous step, from Recursive Feature Elimination (RFE). The goal is to further refine the feature set by iteratively adding features that improve the model's performance.

4. Update selected features:-

   Update the **selected_features** variable with the features selected by both the initial step (possibly RFE) and the additional features selected by SFS. The **sfs.k_feature_idx_** attribute contains the indices of the features selected by SFS, and you use these indices to extract the corresponding column names from the original feature set.

5. retrain the Random Forest model using only the selected features obtained from the combination of RFE and SFS. This helps improve model efficiency and interpretability.

```
plt.figure(figsize=(8, 6))
sns.scatterplot(x='GENE669X', y='GENE598X', hue=Lymphoma_class, data=Lymphoma)
plt.title('GENE669X and GENE598X effect on cancer type')
plt.xlabel('GENE669X')
plt.ylabel('GENE598X')
plt.legend(title='Class')
plt.show()
```



There is a positive relation between genes and cancer type.

Dimensionality Reduction

Principal Component Analysis (PCA):

This code performs dimensionality reduction using PCA (Principal Component Analysis) with 6 components on a dataset and then applies three different classification algorithms (Support Vector Machine - SVM, K-Nearest Neighbors - KNN, and Neural Network - NN) on the transformed data.

1. PCA Initialization and Transformation:
   - PCA is initialized with the argument n_components=6, indicating that it should retain 6 principal components.
   - The 'class' column is dropped from the dataset 'Lymphoma', and the remaining columns are stored in the variable X.
   - The 'class' column is assigned to the variable y.
   - The fit_transform() method is used to perform PCA on X, resulting in a transformed dataset stored in the variable components.
2. Explained Variance Ratio:
   - pca.explained_variance_ratio_ retrieves the variance explained by each of the selected principal components.
3. Dimensions of Transformed Data:
   - components.shape returns the dimensions of the transformed data.
4. Train-Test Split and Classification:
   - The transformed components along with the target y are split into training and testing sets using train_test_split() with a test size of 30% and a random state of 42.
   - Three classifiers (SVM, KNN, NN) are initialized and trained using the training data (X_train, y_train) obtained from the PCA-transformed dataset.
   - The score() method is used to calculate the accuracy of each classifier on the test set (X_test, y_test).

Code:

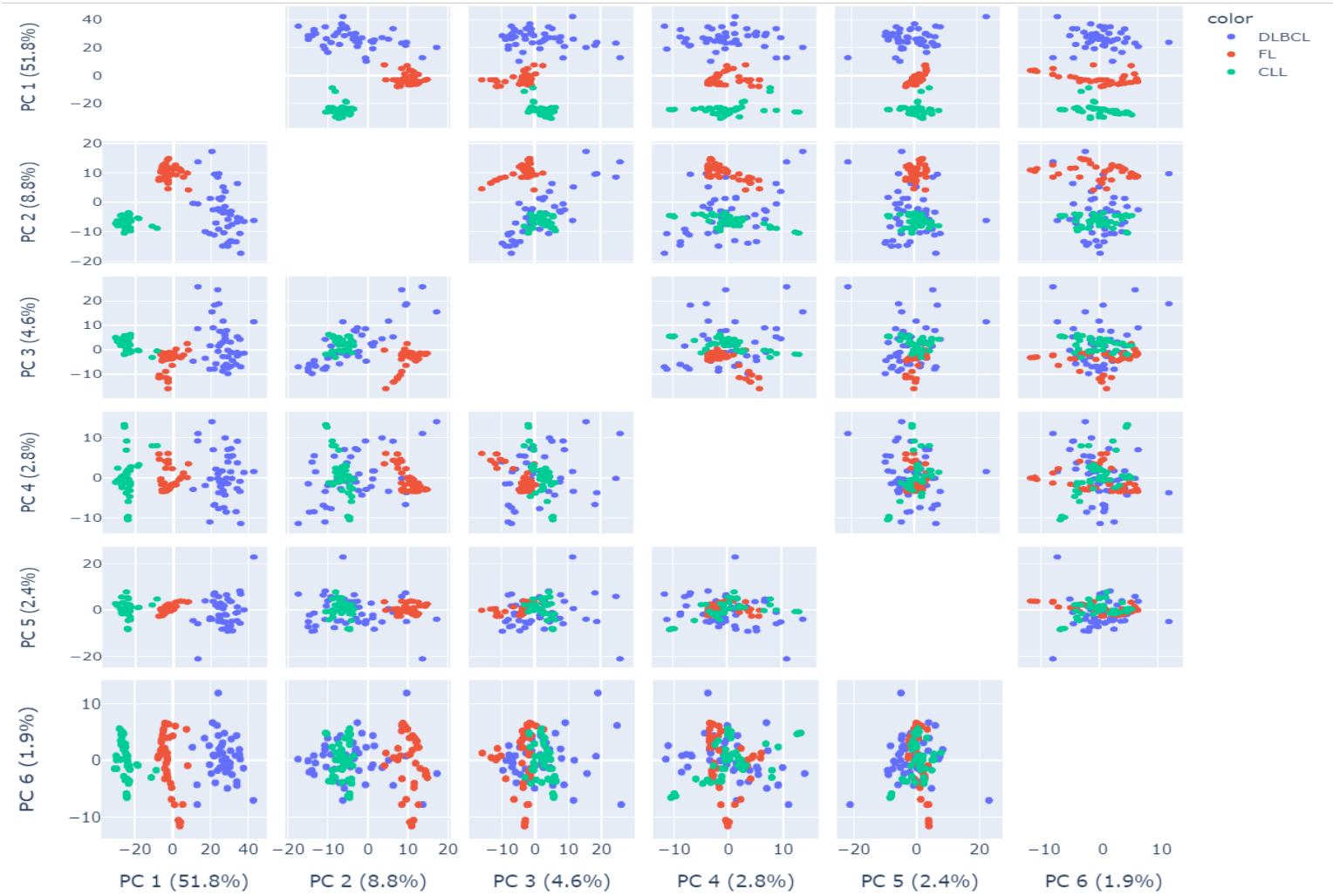
```
pca= PCA(n_components=6)
X = Lymphoma.drop('class', axis=1)
y = Lymphoma['class']
components = pca.fit_transform(X)

explained_variance = pca.explained_variance_ratio_
print("Explained Variance Ratio: ",explained_variance)

print("New data is :  (",components.shape[0],") rows, (",components.shape[1],") columns")

X_train, X_test, y_train, y_test = train_test_split(components, y, test_size=0.3, random_state=42)
svm= SVC()
svm.fit(X_train, y_train)
svm_accuracy = svm.score(X_test, y_test)
print("Support Vector Machine (SVM):", svm_accuracy)

knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
knn_accuracy = knn.score(X_test, y_test)
print("K-Nearest Neighbors (KNN):", knn_accuracy)

nn = MLPClassifier(max_iter = 200)
nn.fit(X_train, y_train)
nn_accuracy = nn.score(X_test, y_test)
print("Neural Network (NN):", nn_accuracy)
```

```
Explained Variance Ratio:  [0.51821125 0.08757781 0.0464762  0.02811014 0.02411818 0.0190363 ]
New data is :  ( 138 ) rows, ( 6 ) columns
Support Vector Machine (SVM): 1.0
K-Nearest Neighbors (KNN): 1.0
Neural Network (NN): 1.0
```

This code snippet creates a scatter matrix plot using Plotly Express, displaying relationships and distributions among the 6 principal components obtained from PCA.

Code:

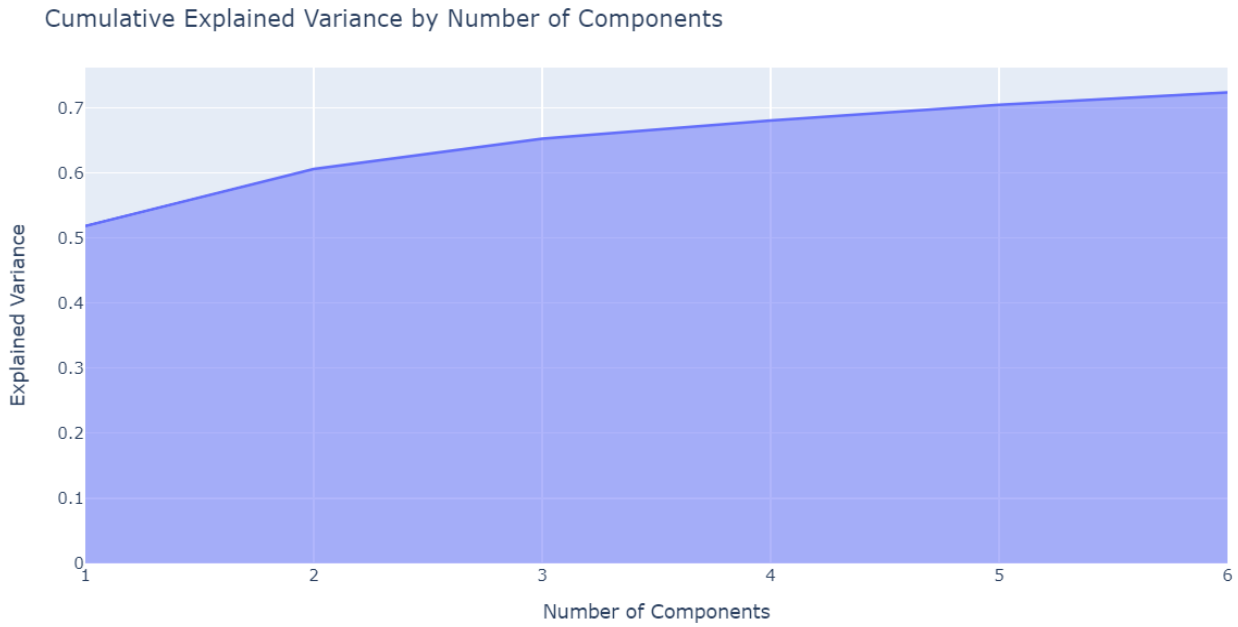
```
labels = {
    str(i): f"PC {i+1} ({var:.1f}%)"
    for i, var in enumerate(pca.explained_variance_ratio_ * 100)
}
fig = px.scatter_matrix(
    components,
    labels=labels,
    dimensions=range(6),
    color=Lymphoma_class
)
fig.update_layout(width=1000, height=1000)
fig.update_traces(diagonal_visible=False)
fig.show()
```



This code snippet generates an area chart that visually represents the cumulative explained variance as the number of components increases in PCA. It helps in determining the number of components needed to retain a significant portion of the variance in the dataset. The plot's x-axis represents the number of components, while the y-axis shows the cumulative explained variance, aiding in the decision of how many components to use for dimensionality reduction while retaining essential information from the original data.

Code:

```python
# Your code to generate the area chart remains the same
exp_var_cumul=np.cumsum(pca.explained_variance_ratio_)
fig = px.area(
    x=range(1, exp_var_cumul.shape[0] + 1),
    y=exp_var_cumul,
    labels={"x": "# Components", "y": "Explained Variance"}
)

# Update layout properties
fig.update_layout(
    title="Cumulative Explained Variance by Number of Components",
    xaxis_title="Number of Components",
    yaxis_title="Explained Variance",
    width=1000,
    height=500
)

fig.show()

```



Gaussian Random Projection:

This code performs dimensionality reduction using Gaussian Random Projection (GRP) and subsequently evaluates three different classification algorithms (Support Vector Machine - SVM, K-Nearest Neighbors - KNN, and Neural Network - NN) on the reduced dataset.

1. Gaussian Random Projection (GRP):

- grp = GaussianRandomProjection(n_components=6, eps=0.9, random_state=42) initializes GRP with parameters:
    - n_components=95: Specifies the number of components to reduce the data.
    - eps=0.9: Sets the randomness factor, controlling the trade-off between preserving pairwise distances and computational efficiency.
    - random_state=42: Sets the random seed for reproducibility.

2. Data Preparation:

- X = Lymphoma.drop('class', axis=1) selects the features (independent variables) by dropping the 'class' column from the 'Lymphoma' dataset.
- y = Lymphoma['class'] assigns the target variable ('class').

3. Dimensionality Reduction:

- y = Lymphoma['class'New_Data2 = grp.fit_transform(X) performs GRP on the feature matrix X, resulting in a new dataset New_Data2 with reduced dimensions.

4. Train-Test Split and Classification:

- train_test_split(New_Data2, y, test_size=0.3, random_state=42) splits the reduced dataset (New_Data2) and the target variable (y) into training and testing sets with a 70-30 split ratio.
- Three classifiers (SVM, KNN, NN) are initialized and trained using the training data (X_train, y_train) obtained from the reduced dataset.
- The score() method is used to calculate the accuracy of each classifier on the test set (X_test, y_test).

5. Model Evaluation:

- svm_accuracy, knn_accuracy, nn_accuracy store the accuracy scores obtained by the Support Vector Machine, K-Nearest Neighbors, and Neural Network classifiers, respectively.
- The accuracy scores for each classifier are printed to the console using print() statements.

Code:

```
grp = GaussianRandomProjection(n_components=95, eps=0.9, random_state=42)

X = Lymphoma.drop('class', axis=1)
y = Lymphoma['class']

New_Data2 = grp.fit_transform(X)

print("New data is :  (",New_Data2.shape[0],") rows, (",New_Data2.shape[1],") columns")

X_train, X_test, y_train, y_test = train_test_split(New_Data2, y, test_size=0.3, random_state=42)
svm= SVC()
svm.fit(X_train, y_train)
svm_accuracy = svm.score(X_test, y_test)
print("Support Vector Machine (SVM):", svm_accuracy)

knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
knn_accuracy = knn.score(X_test, y_test)
print("K-Nearest Neighbors (KNN):", knn_accuracy)

nn = MLPClassifier(max_iter = 200,random_state=42)
nn.fit(X_train, y_train)
nn_accuracy = nn.score(X_test, y_test)
print("Neural Network (NN):", nn_accuracy)
```

```
New data is :  ( 138 ) rows, ( 95 ) columns
Support Vector Machine (SVM): 0.9761904761904762
K-Nearest Neighbors (KNN): 0.9761904761904762
Neural Network (NN): 1.0
```

PCA favored due to its perfect accuracy with a significantly lower number of dimensions.

Clustering After PCA

1- K-Means

K-Means is a popular clustering algorithm used in machine learning and data analysis. Its primary objective is to partition a dataset into distinct, non-overlapping subgroups or clusters

Wcss And Elbow Method:

to determine the optimal number of clusters for a K-means clustering algorithm. We employ the Within-Cluster Sum of Squares (WCSS) metric as a criterion to identify the point of inflection in the curve, also known as the "elbow," which indicates the optimal number of clusters.

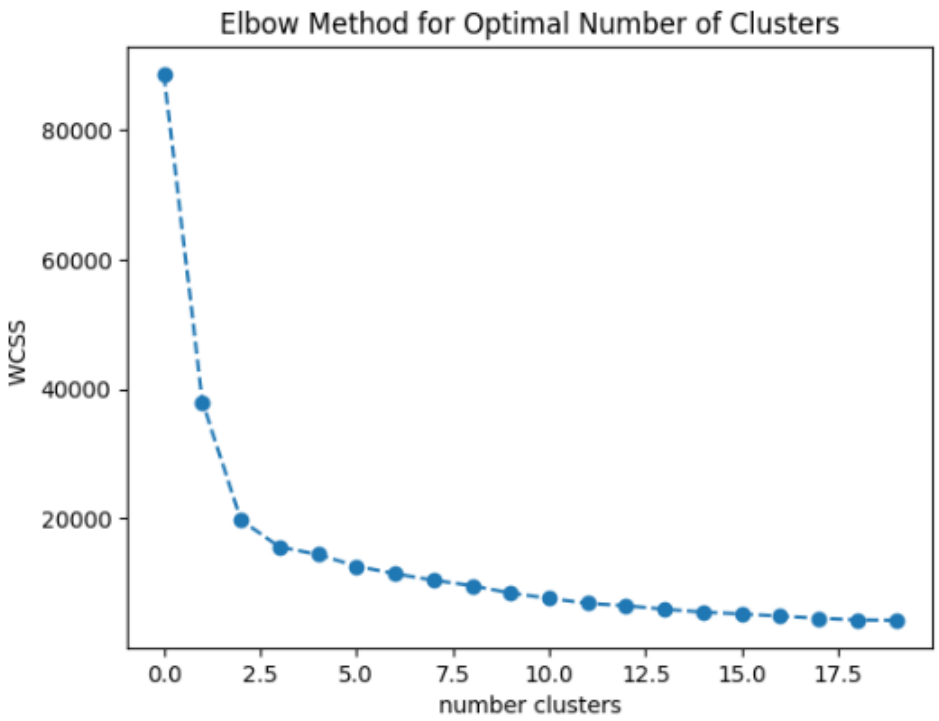
Elbow Method: The Elbow Method involves plotting the WCSS against the number of clusters. The resulting curve is analyzed to identify the point where the rate of decrease in WCSS significantly slows down, resembling an elbow. This point indicates the optimal number of clusters, as it balances the desire for compact clusters (low WCSS) with the goal of not over-segmenting the data.

Finally, we visualize the the elbow plot to determine the optimal number of clusters for kmeans ,The Elbow Method plot exhibits a clear elbow at a cluster number of 3. This suggests that dividing the data into three clusters strikes a balance

Code and output:

```
# to determine the optimal number of clusters in a K-means we use WCSS (Within Cluster Sum of Squares)
wcss=[]
for i in range(1,21):
  kmeans=KMeans(n_clusters=i)
  kmeans.fit(components)
  wcss.append(kmeans.inertia_)
```

```
# we use the Elbow-method to detrmine that the optimal number of clusters is 3
plt.plot(wcss,marker='o',linestyle='--')
plt.title('Elbow Method for Optimal Number of Clusters')
plt.xlabel('number clusters')
plt.ylabel('WCSS ')
plt.show()
```



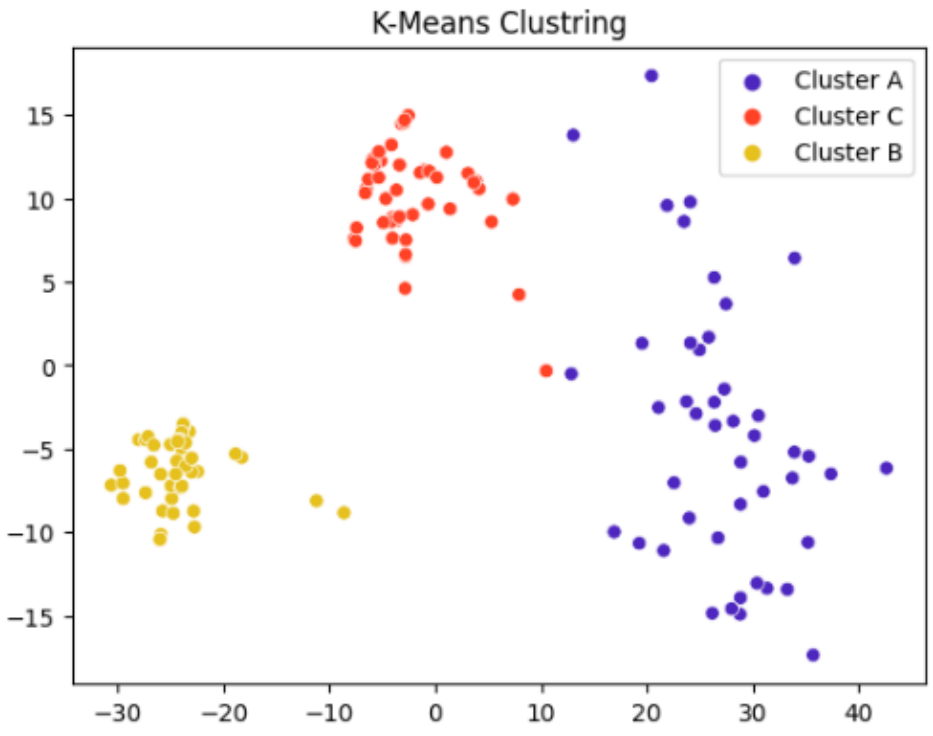
K-Means Clustering: The K-Means algorithm is employed with a predetermined number of clusters, in this case, set to 3. The algorithm is fitted to the 'components' dataset, PCA was applied before the clustering, partitioning it into three clusters based on the inherent patterns in the data.
The default cluster labels assigned by K-Means (0, 1, 2) are renamed for clarity and interpretation, then a scatter plot is generated to visualize the clusters

Code and output:

```
kmeans = KMeans(n_clusters=3)
kmeans.fit(components)
```

```
# renaming the labels
kmeans_labels = kmeans.labels_
kmeans_label_mapping  = {0: 'Cluster A', 1: 'Cluster B', 2: 'Cluster C'}
kmeans_renamed_labels = np.array([kmeans_label_mapping [label] for label in kmeans_labels])
```

```
sns.scatterplot(x=components[:, 0],y= components[:, 1],hue=kmeans_renamed_labels,
                palette='CMRmap').set(title='K-Means Clustring')
plt.show()
```



2- Hc clustering:

Hierarchical clustering (HC) is a method of cluster analysis that builds a hierarchy of clusters. The term "hierarchical" refers to the fact that the method organizes data into a tree-like structure, commonly known as a dendrogram. The hierarchy is created through a series of nested clusters, where at the beginning, each data point is treated as a separate cluster, and then these clusters are successively merged into larger clusters based on their similarity.

There is two main types of Hierarchical clustering Agglomerative Hierarchical and Clustering and Divisive Hierarchical Clustering ,we used **Agglomerative Hierarchical** method in this implementation:

Bottom-Up Approach: Start with each data point as a separate cluster. Iteratively merge the two closest clusters until only a single cluster remains.

Linkage Methods:

- Complete Linkage: The distance between two clusters is defined as the maximum distance between their individual members.

- Single Linkage: The distance between two clusters is defined as the minimum distance between their individual members.

- Average Linkage: The distance between two clusters is defined as the average distance between their individual members. Ward's Method: Minimizes the variance within each cluster.

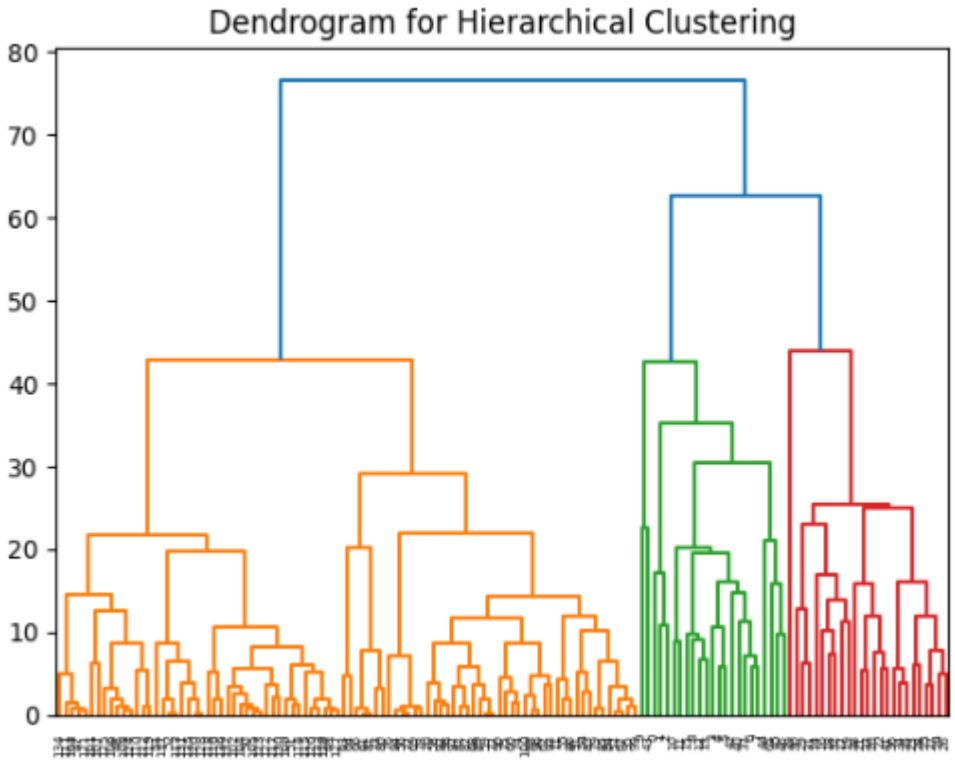
In this example we used **complete linkage method**

Dendrogram for Optimal Number of Clusters: The 'components' dataset is subjected to hierarchical clustering using the complete linkage method, and a linkage matrix is computed. A dendrogram is plotted to visually inspect the structure and determine the optimal number of clusters based on the hierarchical structure of the data which is 3 clusters.

The default cluster labels assigned by HC (0, 1, 2) are renamed for clarity and interpretation, then a scatter plot is generated to visualize the clusters

Code and output:

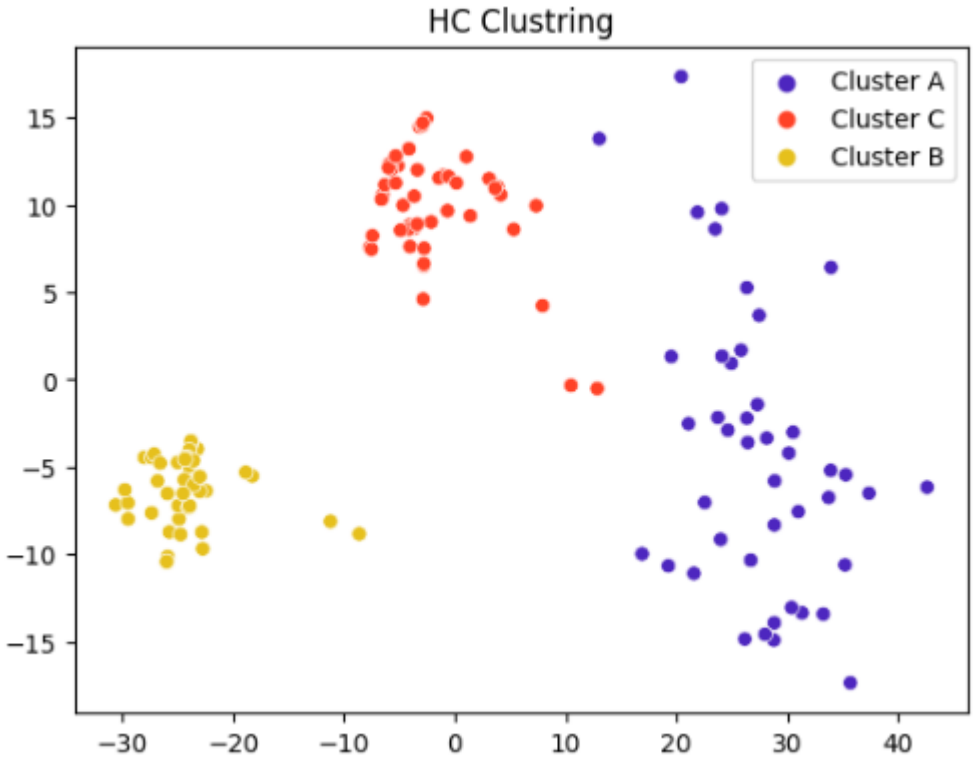
```
# Using complete linkage
linkage_matrix = linkage(components, method='complete')

# Plot the dendrogram to detrmine optimal num of clusters
dendrogram(linkage_matrix)
plt.title("Dendrogram for Hierarchical Clustering")
plt.show()
```



```
#Agglomerative Clustering
hc = AgglomerativeClustering(n_clusters=3)
hc = hc.fit_predict(components)
```

```
# renaming the labels
hc_label_names = {0: 'Cluster A', 1: 'Cluster B', 2: 'Cluster C'}
hc_named_labels = [hc_label_names[label] for label in hc]
```

```
sns.scatterplot(x=components[:, 0],y= components[:, 1],hue=hc_named_labels,
                palette='CMRmap').set(title='HC Clustring')
plt.show()
```



3-DBSCAN Clustering:

DBSCAN, which stands for Density-Based Spatial Clustering of Applications with Noise, is a popular clustering algorithm used in machine learning and data analysis. Unlike traditional clustering algorithms (such as K-Means), DBSCAN does not require a predefined number of clusters and can identify clusters of arbitrary shapes. It is particularly useful for datasets with irregular shapes and varying densities.

Parameters:

Eps (epsilon): The maximum distance between two data points for one to be considered in the neighborhood of the other.

Min_samples: the fewest number of points required to form a cluster.

Challenges:

- Sensitivity to the choice of parameters (Eps and Min_samples).
- Difficulty handling data with varying densities.

determining the optimal value of the epsilon:

The epsilon parameter is the most important parameter in the DBSCAN, by utilizing the Nearest Neighbors approach. The Nearest Neighbors method is employed to calculate the distances to the k-nearest neighbors for each data point, providing insights into the density distribution of the dataset.

Nearest Neighbors Calculation: The Nearest Neighbors algorithm is applied to the 'components' dataset with a specified number of neighbors (k = 5). Distances to the k-nearest neighbors for each data point are computed and sorted.

Nearest Neighbors Plot: A plot is generated to visualize the distances to the k-nearest neighbors. The plot helps identify an appropriate value for the epsilon parameter in DBSCAN by examining the point of inflection or "knee" in the plot.

Results: The Nearest Neighbors plot exhibits a pattern where distances to the nearest neighbors are sorted in ascending order. By analyzing the plot, the point where the curve shows a significant change, often referred to as the "knee" or "elbow," provides insights into an appropriate choice for the epsilon parameter, after many trail and error we determine that an epsilon of 14 and a minimum number of samples 13 is gives us the best silhouette score

DBSCAN Clustering:

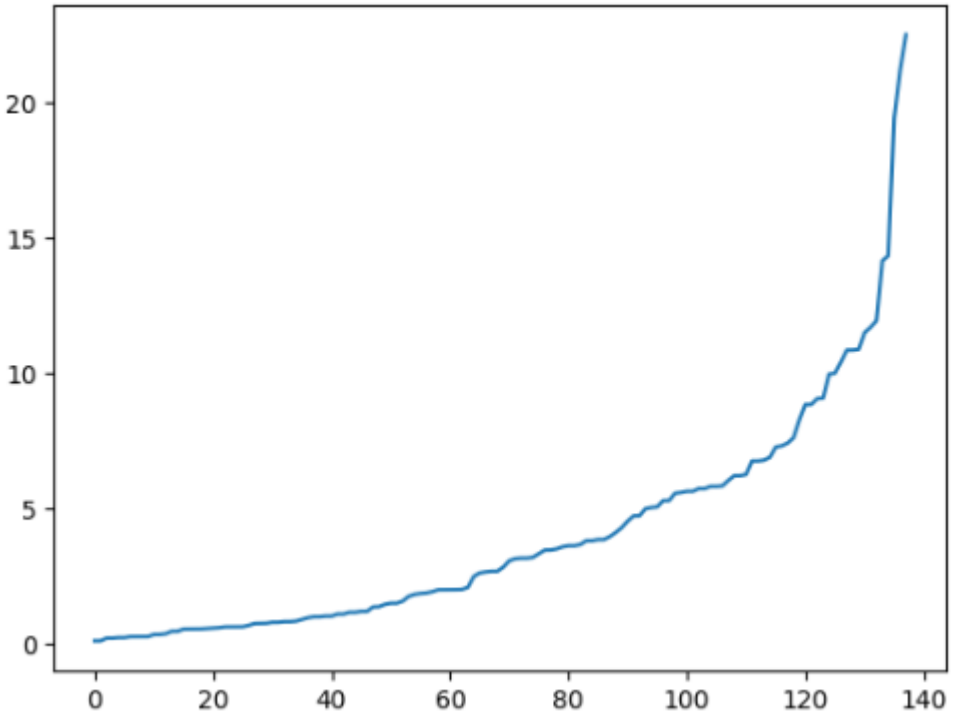
- DBSCAN is applied to the 'components' dataset with the determined optimal values for eps and min_samples.
- Each data point is assigned to a cluster, and noise points are labeled accordingly.

The default cluster labels assigned by DBSCAN (0, 1, 2) are renamed for clarity and interpretation A scatter plot is generated to visually represent the 3 cluters determined by the eps value and noise points

Code and Output:

```
# we use Nearest Neighbors to detrmine the best value for epsilon which is the most important parameter for dbscan

neigh=NearestNeighbors(n_neighbors=5)
nbrs = neigh.fit(components)
distances , indices = nbrs.kneighbors(components)
```

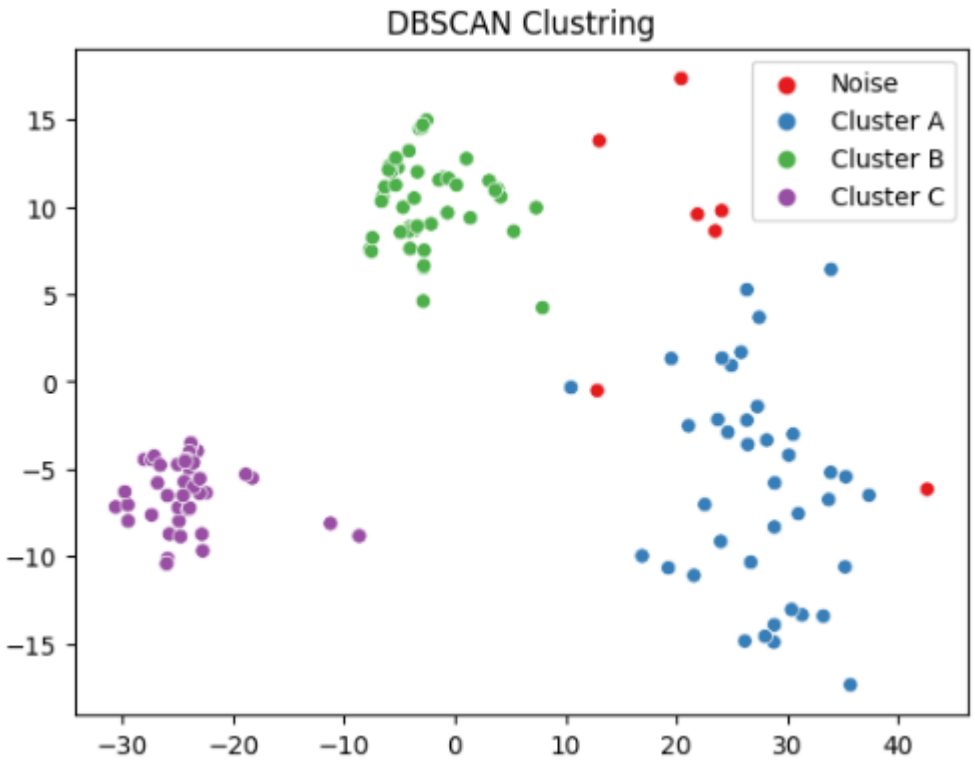
```
distances=np.sort(distances , axis=0 )
distances=distances[:,1]
plt.plot(distances)
```



```
dbscan = DBSCAN(eps=14, min_samples=13) #this is the best values based on the sillhouette score
dbscan = dbscan.fit_predict(components)
```

```
# renaming the lables
dbscan_label_names = {0: 'Cluster A', 1: 'Cluster B', 2: 'Cluster C', -1: 'Noise'}
dbscan_named_labels = [dbscan_label_names[label] for label in dbscan]
```

```
sns.scatterplot(x=components[:, 0],y= components[:, 1],hue=dbscan_named_labels,
                palette='Set1').set(title='DBSCAN Clustring')
plt.show()
```



silhouette score:

The silhouette score is a metric used to measure the quality of clusters in a clustering algorithm. It quantifies how well-separated the clusters are and provides an indication of the appropriateness of the number of clusters (K) chosen for a specific dataset. The silhouette score ranges from -1 to 1, where a higher score indicates better-defined clusters.

Interpretation of Silhouette Score:

- A silhouette score close to +1 indicates that the data point is well-matched to its own cluster and poorly matched to neighboring clusters, suggesting a good clustering.
- A silhouette score around 0 indicates overlapping clusters, where the data point could be assigned to either of the neighboring clusters.
- A silhouette score close to -1 indicates that the data point is probably placed in the wrong cluster.

Code and output:

K-means silhouette score

```
# kmeans silhouette score
# The Silhouette score is a metric used
# to evaluate how good clustering results are in data clustering
kmeans_silhouette = silhouette_score(components, kmeans_labels)
print("K-means Silhouette Score:", kmeans_silhouette)
```

```
K-means Silhouette Score: 0.5510600508418326
```

HC Clustering silhouette score:

```
# hc silhouette score
hc_silhouette = silhouette_score(components, hc)
print("Hierarchical Clustering Silhouette Score:", hc_silhouette)
```

```
Hierarchical Clustering Silhouette Score: 0.5468573300911637
```

DBSCAN Clustering silhouette score:

```
# dbscan silhouette score

dbscan_silhouette = silhouette_score(components, dbscan)
print("DBSCAN Silhouette Score:", dbscan_silhouette)
```

```
DBSCAN Silhouette Score: 0.5433408647883555
```

The three clustering techniques almost give the same .54 score, a silhouette score of 0.54 is generally considered quite good. It indicates a well-defined separation between clusters.

Plotting the 3 techniques together for comparison

```
# plotting all for comparison

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(15, 5))

# DBSCAN Clustering
sns.scatterplot(x=components[:, 0], y=components[:, 1], hue=dbscan_named_labels, palette='Set1', ax=axes[0])
axes[0].set_title('DBSCAN Clustering')

# HC Clustering
sns.scatterplot(x=components[:, 0], y=components[:, 1], hue=hc_named_labels, palette='CMRmap', ax=axes[1])
axes[1].set_title('HC Clustering')

# K-means Clustering
sns.scatterplot(x=components[:, 0], y=components[:, 1], hue=kmeans_renamed_labels, palette='CMRmap', ax=axes[2])
axes[2].set_title('K-means Clustering')

fig.suptitle('Cluster Analysis Comparison')

# Adjust spacing between subplots
plt.tight_layout()

plt.show()
```