A. Implementation

A.1. Pre-Processing

One Hot Encoding

One-Hot-Encoding is used to turn categorical data into numerical, this makes implementing logistic regression easier. Codewise, if we were to keep these values categorical, we would consistently get non-float variable errors. We need to convert these values into a proper format so they can be used successfully.

As mentioned before, our categorical variables come from column 'famhist', these values are 'absent' and 'present'. We will use one-hot-encoding to build two columns () in place of this categorical column. We will denote '0' for 'absent' and '1' for 'present' in the famhist_Present column and vice versa for the famhist_Absent column as shown below. To produce one-hot-encoding, we use the pandas's function get_dummies().

```
c1 = ['famhist']
cx = df1[c1] # Features
print(cx)

fam_en = pd.get_dummies(df1.famhist, prefix='famhist')
print(fam_en)
```

```
     famhist
0    Present
1     Absent
2    Present
3    Present
4    Present
..       ...
457   Absent
458   Absent
459   Absent
460   Absent
461  Present

[462 rows x 1 columns]
     famhist_Absent  famhist_Present
0                 0                1
1                 1                0
2                 0                1
3                 0                1
4                 0                1
..              ...              ...
457               1                0
458               1                0
459               1                0
460               1                0
461               0                1

[462 rows x 2 columns]
```

Splitting Dataset and Normalization

```
X = df_final.iloc[:, 0:10]
y = df1.iloc[:, -1]

from sklearn.model_selection import train_test_split
xTrain, xTest, yTrain, yTest = train_test_split(X, y, test_size = 0.25, random_state = 0)

X_train = xTrain.iloc[:,0:10].apply(lambda x: (x-x.mean())/ x.std(), axis=0)
X_test = xTest.iloc[:,0:10].apply(lambda x: (x-x.mean())/ x.std(), axis=0)

print("Size of X training set: " +str(len(X_train)))
print("Size of X testing set: " +str(len(X_test)))
print("Size of y training set: " +str(len(y_train)))
print("Size of y testing set: " +str(len(y_test)))
print("\n")
print("X_train Data: \n" + str(X_train))
```

```
Size of X training set: 346
Size of X testing set: 116
Size of y training set: 346
Size of y testing set: 116


X_train Data:
     famhist_Present       sbp   tobacco  ...   obesity   alcohol       age
247          1.17567  1.031846  0.055945  ... -0.228276 -0.685001  1.154855
7            1.17567 -1.170120  0.072723  ... -0.696744 -0.394503  1.021484
360          1.17567 -0.269316  1.796615  ...  0.945235  0.244417  1.288226
383         -0.84812 -0.469494 -0.237662  ... -1.120707 -0.685001 -0.779026
22           1.17567  0.631489 -0.720016  ... -0.338366 -0.685001  0.488000
..               ...       ...       ...  ...       ...       ...       ...
323          1.17567  0.131042  1.020654  ...  0.568119 -0.204729  1.421597
192          1.17567 -1.770656 -0.699044  ... -0.584312 -0.595949 -0.245542
117         -0.84812  0.831668 -0.636128  ... -1.284671 -0.685001 -0.045485
47           1.17567 -1.070031 -0.382368  ...  0.919469 -0.529377 -0.645655
172         -0.84812 -0.669673 -0.782932  ...  0.001273 -0.284702  0.287943

[346 rows x 10 columns]
```

Here, we split our dataset with the help of the train_test_split library from sklearn. We split our features (X dataset) into test and train sets. Test set will take 25% of the original dataset and the train set will take 75%. We split the target (y dataset) similarly into y_test and y_train. These train and test sets are needed to make sure that once we are done training our model, our model is able to generalize to new data. Training the model on the data stores information learned from the data. The model learns from the relationship between x_train and y_train.

Once the train and test sets are sorted, we normalize the features within X_train and X_test. The normalized dataset of X_train is displayed in the image above, X_test normalized can be found by simply running "print(X_test)".

B. Implementation

B.1. Part 1

Logistic Regressions and Accuracy

```
model = LogisticRegression()
model.fit(X_train, y_train)
pred = model.predict(X_test)
score = model.score(X_test, y_test)
parameters = model.coef_

print("Coefficients of Model: \n" + str(parameters))
print("\n")
print("Accuracy of Model: " + str(score*100))
```

```
Coefficients of Model:
[[ 0.17620596  0.0318447   0.41934621  0.23297107  0.20656574 -0.17620596
   0.50230761 -0.24665665 -0.05790607  0.72821719]]


Accuracy of Model: 74.13793103448276
```

To predict the logistic regression of our model, we use the logistic regression library sklearn. We run logistic regression on the test set based on the model we trained on the train set. The weights and accuracy of this model can be found in the image above. The accuracy of our logistic regression model is approximately 74.13%

Principal Component Analysis

Principal Component Analysis (PCA) is a statistical procedure and a linear dimensionality reduction technique that can be useful when working with large datasets. It projects data from high dimensional space into lower dimensional space. It removes the parts that have lower variation and keeps the parts that have high variance.

In this section, we discover principal components for all of our features (10 features). The components have both direction and magnitude. The first principal component will hold the most variance within the data. Here, we build the PCA model for both X_train and X_test with the help of sklearn and run it over 10 components. The implementation of this model and outputs for both X_train PCA and X_test PCA is given below.

```python
from sklearn import preprocessing

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

from sklearn.decomposition import PCA

pca = PCA()
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)

explained_variance = pca.explained_variance_ratio_

from sklearn.decomposition import PCA

pca = PCA(n_components=10)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
#print(pd.DataFrame(X_train).sample(5))

X_pca = pd.DataFrame(X_train, columns=['PC1','PC2', 'PC3', 'PC4', 'PC5', 'PC6', 'PC7', 'PC8', 'PC9', 'PC10']) # PC=principal component
X_pca
```

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10
0	1.399114	-0.805093	0.962477	-1.735514	-0.286959	0.795945	-0.097598	-0.276041	0.024121	5.216800e-15
1	-0.143066	-2.058010	0.118136	0.007922	1.450472	0.217116	-0.463066	-0.884720	1.005516	-3.947405e-17
2	2.689873	-0.601413	0.241692	0.216637	1.177743	-0.342070	-0.831735	0.313680	0.165402	-2.531579e-17
3	-1.759899	0.398165	-0.630327	0.325251	0.770611	0.436120	0.287451	-0.208859	-0.490569	1.224830e-17
4	1.433292	-1.292658	-0.883583	-0.323215	0.000168	0.989445	0.524204	-0.765157	-0.771160	-5.375280e-17
...	...	...	...	...	...	...	...	...	...	...
341	2.090629	-0.728337	0.624713	-0.531909	0.368032	0.234667	-1.165461	-0.127684	0.078204	-2.299703e-17
342	-1.076109	-2.427517	-0.319251	-0.457257	0.543620	-0.433154	-0.762983	-0.406929	0.330784	9.549679e-18
343	-0.977532	1.662393	1.407279	-3.580305	-0.734275	-0.753981	1.599961	0.088053	-0.310473	-8.089102e-17
344	0.775608	-1.571449	-1.498426	-0.635174	0.247463	-1.009254	0.357885	0.714851	0.192353	3.295617e-17
345	-0.250253	1.132299	-0.760666	-0.561238	-0.415405	-0.696859	-0.358681	-0.892138	-0.657076	2.080715e-18

346 rows × 10 columns

```python
X_test_pca = pd.DataFrame(X_test, columns=['PC1','PC2', 'PC3', 'PC4', 'PC5', 'PC6', 'PC7', 'PC8', 'PC9', 'PC10']) # PC=principal component
X_test_pca
```

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10
0	2.496314	-0.411754	-0.103013	-2.474661	-0.498351	0.360040	1.673229	-0.694764	-0.330367	-2.584955e-16
1	0.771744	-1.375687	0.945712	-0.021182	0.434031	-0.170992	-1.103808	0.593545	-0.237165	-1.485351e-16
2	0.685781	1.780066	0.594393	-0.636428	0.128477	0.283625	1.130921	-0.451430	-0.004878	1.920427e-16
3	1.515256	-1.145318	0.479191	-0.668609	0.726787	0.218719	0.956316	-0.832895	0.300229	-1.422454e-16
4	-0.603868	0.924818	-0.100839	0.642580	0.567444	0.366439	0.083785	-0.004643	-0.015264	2.520011e-16
...	...	...	...	...	...	...	...	...	...	...
111	-1.038716	-2.326872	-0.440755	-0.775284	-0.063295	-0.474260	-0.272786	0.395643	0.077083	-2.356983e-16
112	0.136173	1.555278	-0.399564	-0.908702	-0.845893	-0.127999	-0.183108	0.573679	-0.231415	1.516006e-16
113	2.282829	-1.024020	-1.243252	0.429115	0.499288	-0.245712	-0.561249	0.489527	-0.567081	-6.298728e-17
114	1.523629	-1.159425	-0.353918	-1.170249	1.287038	-1.226423	1.149305	-0.365413	-0.191991	-8.895715e-17
115	2.130392	-0.401833	0.825889	-2.043531	-1.163204	0.092539	-0.394252	0.537820	-0.202076	-2.525841e-16

116 rows × 10 columns

PC Variation and Accuracy

In order to find how many PCA's explain a certain amount of variation, we calculate the variation of each PC and then the ratio.The variance ratio shows the variance each component holds after projecting the data to a lower dimensional space.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=10)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
#print(pd.DataFrame(X_train).sample(5))

X_pca = pd.DataFrame(X_train, columns=['PC1','PC2', 'PC3', 'PC4', 'PC5', 'PC6', 'PC7', 'PC8', 'PC9', 'PC10']) # PC=principal component

variance = pca.explained_variance_ratio_ #calculate variance ratios
print("Variance of PCs: \n" +str(variance))
print("\n")
var=np.cumsum(np.round(pca.explained_variance_ratio_, decimals=3)*100)
print("Variance Ratios of PCs: " + str(var)) #cumulative sum of variance explained with [n] features

model_pca = LogisticRegression()
model_pca.fit(X_train, y_train)
pred = model.predict(X_test)
score_pca = model.score(X_test, y_test)

print("\n")
print("Accuracy of PCA Model: " + str(score_pca*100))
```

```
Variance of PCs:
[3.03303074e-01 1.83259998e-01 1.23329699e-01 1.02035896e-01
 8.65859714e-02 7.64184112e-02 6.30279309e-02 4.60522618e-02
 1.59867581e-02 8.02752969e-33]


Variance Ratios of PCs: [30.3 48.6 60.9 71.1 79.8 87.4 93.7 98.3 99.9 99.9]


Accuracy of PCA Model: 72.41379310344827
```

Looking at the results above, we can see that the first PC holds 30.3% of the information while PC9 and PC10 hold 99.9%. The accuracy of our PCA model is approximately 72.41%, this result isn't much different from our logistic regression model. We can conclude that our PCA model for all 10 features results in a lower accuracy, hence a better model would be the logistic regression model.

Question 1: How many PCs explain more than 90%?

As mentioned above and as seen from the image in section **PC Variation and Accuracy**, we can see that only four PCs explain more than 90%. These four PCs include PC7, PC8, PC9 and PC10, explaining 93.7%, 98.3%, 99.9% and 99.9%, respectively.

Question 2: How much variance is explained based on two first PCs?

Based on the first two PCs, only 30.3% variance is explained for the first PC and 48.6% of variance is explained for the second PC.

PCA Model for PCs that Explain More than 90%

Now let's evaluate a new model that considers PCs that explain more than 90% of variance. Here, we run the model for 7 components, seeing as the first PC that explains more than 90% is component 7. This results in accuracy of 70.69%. This result is less than our logistic regression model and our PCA model for all features.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=7)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
print(pd.DataFrame(X_train).sample(5))

variance = pca.explained_variance_ratio_ #calculate variance ratios
print("Variance of PCs: \n" +str(variance))
print("\n")
var=np.cumsum(np.round(pca.explained_variance_ratio_, decimals=3)*100)
print("Variance Ratios of PCs: " + str(var)) #cumulative sum of variance explained with [n] features

from sklearn.ensemble import RandomForestClassifier

classifier = RandomForestClassifier(max_depth=2, random_state=0)
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

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

cm = confusion_matrix(y_test, y_pred)
print('Accuracy: ' + str(accuracy_score(y_test, y_pred)))
```

```
            0         1         2         3         4         5         6
306  1.466684 -0.961390 -0.419088 -1.426499 -0.206153 -0.535805  0.171103
278  2.008823 -0.867651 -0.689184 -0.282048  0.015125  0.461316 -0.461441
307 -2.757321 -0.072458 -0.707367  0.561826  0.165801  0.382457  0.315384
179 -2.815245  0.132345 -0.179561 -0.430387 -0.251607 -0.074127  0.015476
291  1.921077 -1.031966 -0.600393 -0.123769  0.061957  0.571343 -0.364106
Variance of PCs:
[0.30330307 0.18326    0.1233297  0.1020359  0.08658597 0.07641841
 0.06302793]


Variance Ratios of PCs: [30.3 48.6 60.9 71.1 79.8 87.4 93.7]
Accuracy: 0.7068965517241379
```

Question 3: Which of these models performs better?

As discussed in the sections before, the logistic regression model (accuracy of 74.13%) performs the best compared to the all PCs and logistic regression model (accuracy of 72.41%) and PCs that explain more than 90% model (accuracy of 70.7%).

B.2. Part 2

K-Means Clustering

Next, we explore K-Means clustering. Using K-Means clustering helps us predict the clusters our features can fall under. Here, we have two classes of clusters under our target value, ‘chd’. These classes are of ‘0’ and ‘1’. Our K-Means cluster here runs through the X_train set and divides the data into clusters. Since we know we have class 0 and class 1 as our respective clusters, our K value will equal 2. Let’s see if the predictions for each point matches the label in our target value, y. The implementation and output of this K-Means array is shown below.

```
from sklearn.cluster import KMeans
X_scaled = X_train
nclusters = 2 # this is the k in kmeans
seed = 0

km = KMeans(n_clusters=nclusters, random_state=seed)
km.fit(X_scaled)

# predict the cluster for each data point
y_cluster_kmeans = km.predict(X_scaled)
print("Cluster Prediction: \n" + str(y_cluster_kmeans))
print("\n")
from sklearn import metrics
score = metrics.silhouette_score(X_scaled, y_cluster_kmeans)
print("Accuracy of K-Means" +str(score))
```

```
Cluster Prediction:
[1 1 1 0 1 1 0 0 1 1 1 0 1 1 0 0 1 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 0 1
 1 1 1 0 1 0 1 0 1 0 0 1 0 1 1 1 1 0 1 1 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1 1 0
 0 1 0 1 0 1 1 0 0 1 1 0 0 0 1 1 0 1 1 0 0 1 0 1 0 1 1 0 1 0 1 1 1 1 0 1 0
 1 1 0 0 0 0 1 0 0 1 1 0 0 1 1 1 0 1 1 1 1 0 0 0 0 1 1 0 0 0 1 1 0 1 1 1 0
 1 0 1 0 1 0 1 1 1 1 1 0 1 0 0 1 1 1 0 1 1 1 1 1 0 0 0 1 0 1 0 0 0 1 1 0 1
 0 1 0 1 0 1 1 1 1 0 1 0 0 1 1 1 0 1 0 0 1 1 1 0 1 1 0 0 0 0 0 0 0 1 1 1 1
 0 1 1 1 1 0 1 0 1 1 1 1 0 1 1 0 1 0 1 1 0 0 1 1 0 1 0 1 1 0 1 0 1 1 1 0 1
 1 1 0 0 0 0 0 1 1 1 1 1 0 1 0 0 1 1 0 1 0 1 0 0 0 1 0 1 0 1 1 0 1 1 1 0 0
 1 0 0 1 1 0 0 0 1 1 1 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 1 0 1 0 1 0 1 1 0 0 1
 0 1 1 1 1 0 0 0 1 0 0 1 0]


Accuracy of K-Means0.19922129418174686
```

The accuracy for our K-Means model is approximately 19.92%. This model performs worse than our PCA and logistic regression models.

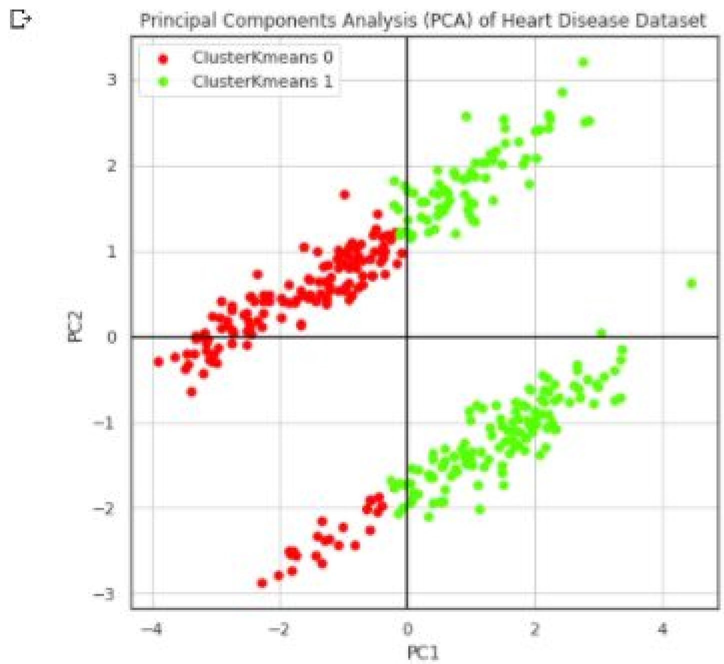
Question 4: How well the dataset is clustered into two groups based on the K-Means algorithm by comparing cluster and class labels.

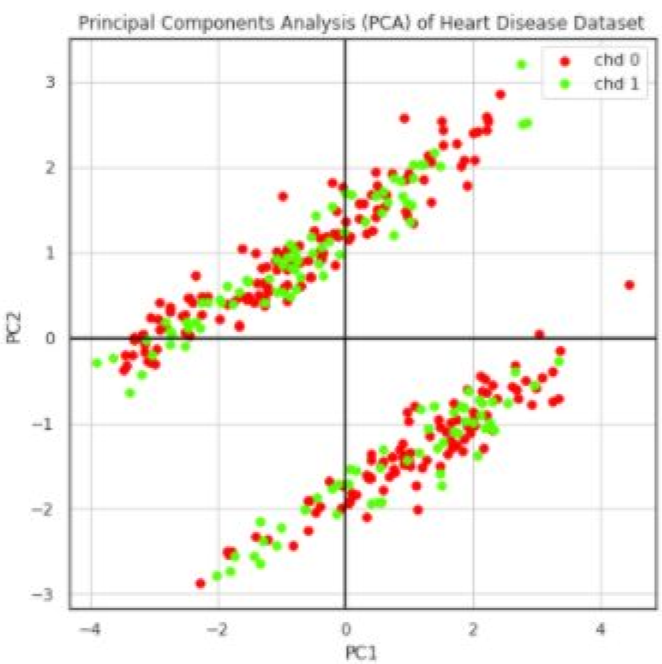
Looking at the accuracy score for the K-Means clustering algorithm, we can conclude that the K-Means does not perform well. Hence, the dataset is not clustered into '0' and '1' classes properly. This makes our predictions through the K-Means algorithm less reliable.

Visualization

Finally, we visualize our models for K-Means and PCA. First, we run K-Means based on the first two PCs and visualize the results. And then we visualize another graph with running a model of PCA based on the first two PCs. The implementation of this procedure is displayed below, the graphs are also displayed below, respectively.

```
df_scores = pd.DataFrame()
df_scores['SilhouetteScore'] = scores
df_scores['chd'] = df_final['chd']

from sklearn.decomposition import PCA

ndimensions = 2

pca = PCA(n_components=ndimensions, random_state=seed)
pca.fit(X_scaled)
X_pca_array = pca.transform(X_scaled)
X_pca = pd.DataFrame(X_pca_array, columns=['PC1','PC2']) # PC=principal component

df_plot = X_pca.copy()
df_plot['ClusterKmeans'] = y_cluster_kmeans
df_plot['chd'] = y # also add actual labels so we can use it in later plots

def plotData(df, groupby):
    "make a scatterplot of the first two principal components of the data, colored by the groupby field"
    fig, ax = plt.subplots(figsize = (7,7))
    cmap = mpl.cm.get_cmap('prism')
    for i, cluster in df.groupby(groupby):
        cluster.plot(ax = ax, # need to pass this so all scatterplots are on same graph
                     kind = 'scatter',
                     x = 'PC1', y = 'PC2',
                     color = cmap(i/(nclusters-1)), # cmap maps a number to a color
                     label = "%s %i" % (groupby, i),
                     s=30) # dot size
    ax.grid()
    ax.axhline(0, color='black')
    ax.axvline(0, color='black')
    ax.set_title("Principal Components Analysis (PCA) of Heart Disease Dataset");

plotData(df_plot, 'ClusterKmeans')
plotData(df_plot, 'chd')
```


Principal Components Analysis (PCA) of Heart Disease Dataset
ClusterKmeans 0
ClusterKmeans 1
PC2
PC1


Principal Components Analysis (PCA) of Heart Disease Dataset
chd 0
chd 1
PC2
PC1

Question 5: Compare the results side by side and describe how well the K-Means algorithm performs after using two PCs.

The K-Means plot displays the clusters that each datapoint was assigned to, these are our predicted values. The PCA model of two PCs displays the actual results within our dataset. Since we already know that the PCA model is more accurate, our K-Means algorithm presents us with wrong cluster classifications.