

Amrita School of Engineering

Bengaluru, India



19AIE205-Python for Machine Learning

PREDICTION OF CARDIOVASCULAR DISEASE

Team Name: Outliers

Team Members:

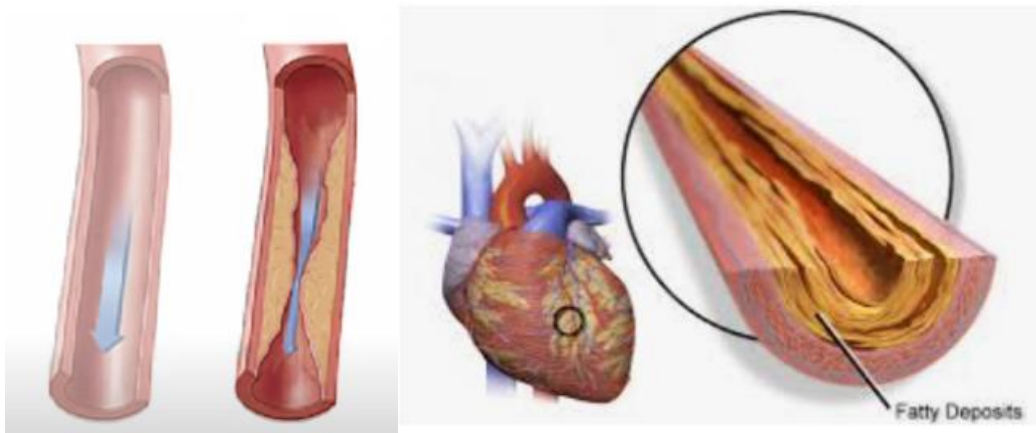
BL.EN.U4AIE19007 - Apoorva Mani

BL.EN.U4AIE19010 – Bhuvanashree Murugadoss

BL.EN.U4AIE19010 – Karna Sai Nikhilesh Reddy

Introduction to Cardiovascular Diseases:

Cardiovascular disease involves abnormalities of the heart and the blood vessels. The lifetime risk of developing significant cardiovascular disease is greater than the likelihood of developing cancer. Our heart is a muscular pump that requires heart arteries to supply oxygen rich blood to keep it going. Coronary heart disease occurs when these blood vessels become narrowed due to a buildup of plaque. The plaque is made up of cholesterol and other substances. This process is called Atherosclerosis.



Atherosclerosis occurs over a life time, and is influenced by risk factors. Some risk factors can't be changed; however, others can be altered through healthy lifestyle choices and medications if needed.

Symptoms of coronary heart disease occur when heart muscle does not get enough oxygen. Some describe this as chest pressure or chest pain that starts over the left chest and can radiate or travel to the left arm, left jaw, back, or sometimes to the right chest. Some patients do not have any chest pain or pressure symptoms but instead have intense shortness of breath or epigastric discomfort that can feel like a bad heart burn, breathlessness, nausea, vomiting or sweats can also sometimes be associated. These symptoms are called angina. There are many treatments for angina, including medications, angioplasty and stents, and bypass surgery. Aggressive risk factor modification is part of every treatment plan. Coronary heart disease can have serious complications including heart attack and even sudden death. Fortunately, developing significant coronary heart disease is largely preventable. Early prevention is best, but you can still improve how you'll do no matter how old you are by reducing your risk factors!

Objective of the project:

What we aim to do is create an efficient classification model that best predicts if a given patient is likely to develop cardio vascular disease or not.

Overview of our approach towards the problem:

1. Perform Exploratory Data Analysis to gain insights on the Data
 - a. Data Cleaning
 - b. Data Summarization: Describe the data and its distributions
 - c. Data Visualization: Create graphical summaries of the data
2. Transform the data for training the different classification algorithms
3. Apply the different Machine Learning Algorithms
 - a. Train, Cross validate, perform appropriate hyperparameter tuning
 - b. Comparative analysis of the accuracy of the models
4. Gain insights from the results obtained

Dataset Description:

Source: <https://www.kaggle.com/sulianova/cardiovascular-disease-dataset>

The dataset consists of 70,000 records of patients data, with 11 features and 1 target variable

There are 3 types of input features:

- Objective: factual information
- Examination: results of medical examination
- Subjective: information given by the patient

Features:

1. Age | Objective Feature | age | int (days)
2. Height | Objective Feature | height | int (cm) |
3. Weight | Objective Feature | weight | float (kg) |
4. Gender | Objective Feature | gender | categorical code |
5. Systolic blood pressure | Examination Feature | ap_hi | int |

-Systolic blood pressure, the top number, measures the force your heart exerts on the walls of your arteries each time it beats

6. Diastolic blood pressure | Examination Feature | ap_lo | int |

-Diastolic blood pressure, the bottom number, measures the force your heart exerts on the walls of your arteries in between beats.

Blood Pressure Category	Systolic mm Hg (upper #)		Diastolic mm Hg (lower #)
Normal	less than 120	and	less than 80
Elevated	120-129	and	less than 80
High Blood Pressure (Hypertension) Stage 1	130-139	or	80-89
High Blood Pressure (Hypertension) Stage 2	140 or higher	or	90 or higher
Hypertensive Crisis (Seek Emergency Care)	higher than 180	and/or	higher than 120

7. Cholesterol | Examination Feature | cholesterol | 1: normal, 2: above normal, 3: well above normal |

Over the years, cholesterol and fat in the blood are deposited in the inner walls of the arteries that supply blood to the heart, called the coronary arteries. These deposits make the arteries narrower, a condition known as atherosclerosis.


	DESIRABLE	BORDERLINE	HIGH RISK
Cholesterol	<200 mg/dl	200-239 mg/dl	240 mg/dl

8. Glucose | Examination Feature | gluc | 1: normal, 2: above normal, 3: well above normal |

The blood glucose level is the amount of glucose in the blood. Glucose is a sugar that comes from the foods we eat, and it's also formed and stored inside the body. It's the main source of energy for the cells of our body, and it's carried to each cell through the bloodstream.

BLOOD GLUCOSE CHART

	Mg/DL	Fasting	After Eating	2-3 hours After Eating
Normal		80-100	170-200	120-140
Impaired Glucose		101-125	190-230	140-160
Diabetic		126+	220-300	200 plus



9. Smoking | Subjective Feature | smoke | binary |

10. Alcohol intake | Subjective Feature | alco | binary |

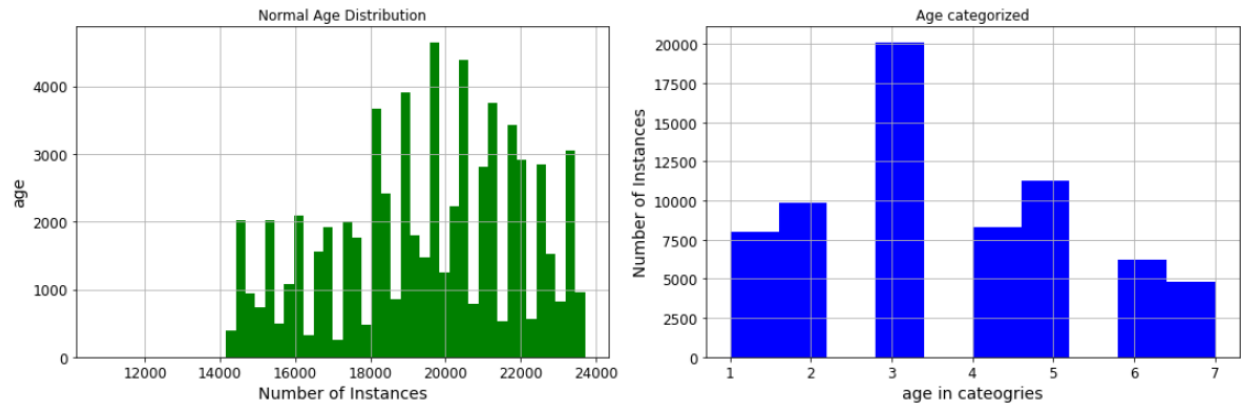
11. Physical activity | Subjective Feature | active | binary |

12. Presence or absence of cardiovascular disease | Target Variable | cardio | binary |

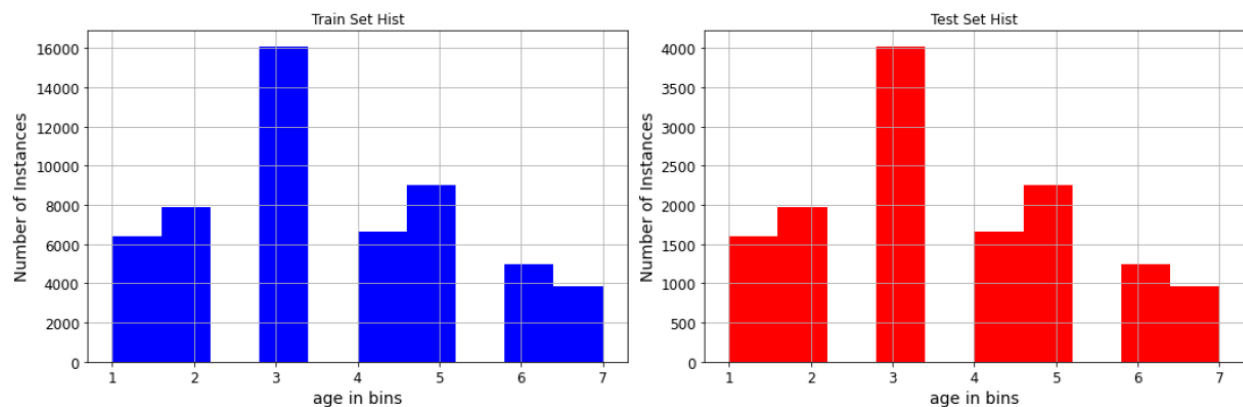
All of the dataset values were collected at the moment of medical examination.

Train Test Split using Stratified Shuffle Split Method

As we can see the distribution below from range 10798 to 24000, if we divide using random train test split method, we might end up having different ratios of categories in test set. So, to proceed further we will divide the age feature into 7 bins and do Stratified Shuffle Split.



After applying the method, we have ended up with same ratio of all the categories. We have divided the actual data into two sets with test ratio as 20%.



Data Transformation Pipeline

This process is used to clean the new data into more efficient data like scaling, column encoding. We have used Pipeline feature in scikit learn to automate new data flow. We made Numerical and Category Pipeline for processing continuous and categorical data. We have combined both these pipelines with Feature Union feature to combine to form new dataset. Implementation for this Transformation Pipeline is given below.

```
class DataFrameSelector(BaseEstimator, TransformerMixin):

    def __init__(self, attribute_names):
        self.attribute_names = attribute_names

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        return X[self.attribute_names].values

target_feature = ['cardio']
cat_attributes = ['cholesterol', 'gluc']
num_attributes = ['age', 'gender', 'height', 'weight', 'ap_hi', 'ap_lo', 'smoke', 'alco', 'active']
new_cat_attributes = ['cholesterol_1', 'cholesterol_2', 'cholesterol_3', 'gluc_1', 'gluc_2', 'gluc_3']

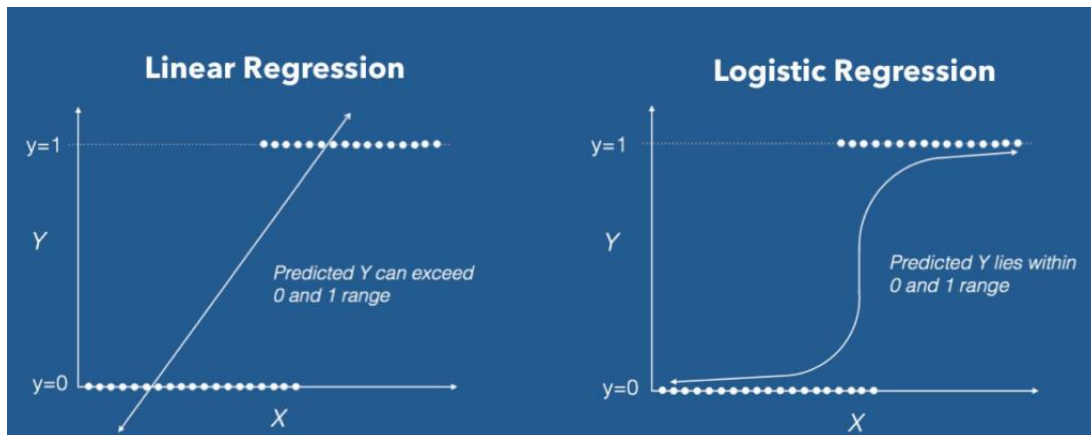
num_pipeline = Pipeline([
    ('selector' , DataFrameSelector(num_attributes)),
    ('imputer' , SimpleImputer(strategy='median')),
    ('std_scaler', StandardScaler())
])

cat_pipeline = Pipeline([
    ('selector' , DataFrameSelector(cat_attributes)),
    ('cat_encoder' , OneHotEncoder(sparse=False))
])

full_pipeline = FeatureUnion(transformer_list=[
    ('Numerical_Pipeline', num_pipeline),
    ('Categorical_Pipeline', cat_pipeline)
])
```

Logistic Regression

Logistic regression (LR) is a statistical method similar to linear regression since LR finds an equation that predicts an outcome for a binary variable, Y, from one or more response variables, X. However, unlike linear regression the response variables can be categorical or continuous, as the model does not strictly require continuous data. To predict group membership, LR uses the log odds ratio rather than probabilities and an iterative maximum likelihood method rather than a least squares to fit the final model.



Implementation:

The model was initially created with default hyper parameters, fit to the training data set and later hyperparameter tuning was done using the Grid Search Model and the accuracy was measured.

‘Penalty’: A regression model that uses L1 regularization technique is called Lasso Regression and model which uses L2 is called Ridge Regression.

‘C’: Inverse of regularization strength; must be a positive float.

```
log_reg=LogisticRegression()
grid_values = {'penalty': ['l1', 'l2'], 'C':[0.001,.009,0.01,.09,1,5,10,25]}
log_reg1 = GridSearchCV(log_reg, param_grid = grid_values,scoring = 'recall')
log_reg1.fit(X_train, Y_train)
```

```
) y_pred = log_reg1.predict(X_test)
  print('Accuracy Score : ' + str(accuracy_score(Y_test,y_pred)))
```

```
› Accuracy Score : 0.6997857142857142
```



```

from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report
print(classification_report(Y_test, y_pred))

```

	precision	recall	f1-score	support
0	0.69	0.74	0.71	6988
1	0.72	0.66	0.69	7012
accuracy			0.70	14000
macro avg	0.70	0.70	0.70	14000
weighted avg	0.70	0.70	0.70	14000

```
print(accuracy_score(Y_test,y_pred_acc))
```

0.6997857142857142

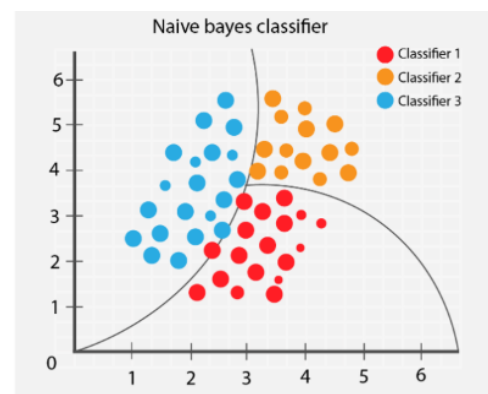
Naive Bayes Classifier

Naive Bayes classifiers calculate the probability of a sample to be of a certain category, based on prior knowledge. They use the Naive Bayes Theorem, that assumes that the effect of a certain feature of a sample is independent of the other features. That means that each character of a sample contributes independently to determine the probability of the classification of that sample, outputting the category of the highest probability of the sample.

$$P(A|B) = \frac{P(B|A) P(A)}{P(B)}$$

using Bayesian probability terminology, the above equation can be written as

$$\text{Posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{evidence}}$$



Implementation:

The model was created fit to the training data set. Grid Search Model was not used for this model since it does not have any hyper parameters.

```

Naive_bayes=GaussianNB()
Naive_bayes.fit(X_train,Y_train)
y_pred=Naive_bayes.predict(X_test)

```

```
[11] from sklearn.model_selection import cross_val_score
      from sklearn.metrics import classification_report
      print(classification_report(Y_test, y_pred))
```

	precision	recall	f1-score	support
0	0.54	0.93	0.69	6988
1	0.76	0.22	0.34	7012
accuracy			0.57	14000
macro avg	0.65	0.57	0.51	14000
weighted avg	0.65	0.57	0.51	14000

```
) print(accuracy_score(Y_test,y_pred))
```

0.5739285714285715

K Nearest Neighbors

K nearest neighbors is a simple algorithm that stores all available cases and classifies new cases based on a similarity measure (e.g., distance functions). KNN has been used in statistical estimation and pattern recognition already in the beginning of 1970's as a non-parametric technique.

Algorithm

A case is classified by a majority vote of its neighbors, with the case being assigned to the class most common amongst its K nearest neighbors measured by a distance function. If $K = 1$, then the case is simply assigned to the class of its nearest neighbor.

Distance functions

Euclidean	$\sqrt{\sum_{i=1}^k (x_i - y_i)^2}$
Manhattan	$\sum_{i=1}^k x_i - y_i $
Minkowski	$\left(\sum_{i=1}^k (x_i - y_i)^q \right)^{1/q}$

It should also be noted that all three distance measures are only valid for continuous variables. In

the instance of categorical variables the Hamming distance must be used. It also brings up the issue of standardization of the numerical variables between 0 and 1 when there is a mixture of numerical and categorical variables in the dataset.

Choosing the optimal value for K is best done by first inspecting the data. In general, a large K value is more precise as it reduces the overall noise but there is no guarantee. Cross-validation is another way to retrospectively determine a good K value by using an independent dataset to validate the K value. Historically, the optimal K for most datasets has been between 3-10. That produces much better results than 1NN.

K Nearest Neighbors Implementation

The model was created with tuned Hyper parameters using Grid Search model, fit to the training data set and its accuracy was measured using the K-Fold Cross Validation technique:

```
hyperparameters={
    "n_neighbors" : [1, 5, 10, 25, 50, 75],
    "p" : [1, 2] # manhattan_distance (L1), and euclidean_distance (L2)
}

grid_search=GridSearchCV(KNeighborsClassifier(),hyperparameters,cv=10, verbose=1, n_jobs=-1)
grid_search.fit(X_train,y_train)

print("tuned hyperparameters :(best parameters) ",grid_search.best_params_)
print("accuracy :",grid_search.best_score_)

Fitting 10 folds for each of 12 candidates, totalling 120 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 34 tasks | elapsed: 1.1min
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed: 4.6min finished
tuned hyperparameters :(best parameters) {'n_neighbors': 75, 'p': 1}
accuracy : 0.7289059087703583

#Training the model with the optimised parameters
knn = KNeighborsClassifier(n_neighbors = 75, p = 1)
knn.fit(X_train, y_train)

#KFold cross-validation for evaluating the model
cv = KFold(n_splits=10, random_state=1, shuffle=True)
scores = cross_val_score(knn, X_train, y_train, scoring='accuracy', cv=cv, n_jobs=-1)

# Reporting the performance
print('Scores -> Mean: %.3f | STD: (%.3f)' % (np.mean(scores), np.std(scores)))

Scores -> Mean: 0.728 | STD: (0.005)
```

Hyper Parameters tuned:

1. "n_neighbors": To determine the value of k
2. "p": To calculate distances, 2 distance metrics that are used which are Euclidean Distance and Manhattan Distance

SVM:

SVM or Support Vector Machine is a linear model for classification and regression problems. It can solve linear and non-linear problems and work well for many practical problems

At first approximation what SVMs do is to find a separating line(or hyperplane) between data of two classes. SVM is an algorithm that takes the data as an input and outputs a line that separates those classes if possible.

According to the SVM algorithm we find the points closest to the line from both the

classes. These points are called support vectors. Now, we compute the distance between the line and the support vectors. This distance is called the margin. Our goal is to maximize the margin. The hyperplane for which the margin is maximum is the optimal hyperplane.

Thus SVM tries to make a decision boundary in such a way that the separation between the two classes (that street) is as wide as possible.

SVM Implementation:

The model was created with tuned Hyper parameters using Grid Search model, fit to the training data set and its accuracy was measured using the K-Fold Cross Validation technique:

```
hyperparameters={
    "C" : [0.1, 1, 25, 50, 75],
    "gamma" : [0.1, 1, 10, 25],
    "kernel" : ['linear']
}

grid_search=GridSearchCV(SVC(), hyperparameters,cv=3, verbose=1, n_jobs=-1)
grid_search.fit(X_train,y_train)

print("tuned hyperparameters :(best parameters) ",grid_search.best_params_)
print("accuracy :",grid_search.best_score_)

#Training the model with the optimised parameters
svm = SVC(C=1, gamma=1, kernel='linear')
svm.fit(X_train, y_train)

#KFold cross-validation for evaluating the model
cv = KFold(n_splits=10, random_state=1, shuffle=True)
scores = cross_val_score(svm, X_train, y_train, scoring='accuracy', cv=cv, n_jobs=-1)

# Reporting the performance
print('Scores -> Mean: %.3f | STD: (%.3f)' % (np.mean(scores), np.std(scores)))
```

Hyper Parameters tuned:

1. "C" : It adds a penalty for each misclassified data point.
2. "gamma" : It controls the distance of influence of a single training point.
3. "Kernel" : It maps the observations into some feature space. Ideally the observations are more easily (linearly) separable after this transformation.

Decision Tree Classifier:

Decision Tree algorithm is a supervised learning algorithm that can be used for solving regression and classification problems. The goal of using a Decision Tree is to create a training model that can use to predict the class or value of the target variable by learning simple decision rules inferred from prior data (training data). In Decision Trees, for predicting a class label for a record we start from the root of the tree. We compare the values of the root attribute with the record's attribute. On the basis of comparison, we follow the branch corresponding to that value and jump to the next node.

Methodology followed to train the Decision Tree Classifier:

The model was created using the default hyper parameters, fit to the training dataset and its accuracy was measured using the K-Fold Cross Validation technique:

```
#Training the model with the default parameters
tree_clf = DecisionTreeClassifier()
tree_clf.fit(train_f,train_t)

#KFold cross-validation for evaluating the model
cv = KFold(n_splits=10, random_state=1, shuffle=True)
scores = cross_val_score(tree_clf, train_f, train_t, scoring='accuracy', cv=cv, n_jobs=-1)

# Reporting the performance
print('Accuracy: %.3f (%.3f)' % (np.mean(scores), np.std(scores)))
```

Accuracy: 0.631 (0.005)

Following the above step, hyper parameter tuning was performed on the Decision Tree Classifier using the Grid Search Method. The hyperparameters in tuned here are:

min_samples_split : The minimum number of samples required to split an internal node.

min_samples_leaf :The minimum number of samples required to be at a leaf node.

max_depth : The maximum depth of the tree.

```
[14]: pipeline = Pipeline(['clf',tree_clf])
      parameters = {
          'clf__max_depth': np.linspace(1,10,10),
          'clf__min_samples_split': (1,2,3,4,5),
          'clf__min_samples_leaf': (1,2,3,4,5)
      }
      grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1)
      grid_search.fit(train_f,train_t)

      grid_search.best_score_
```

[14]: 0.7287809523809524

The model is then created and trained using the optimized hyperparameters, to result in an improvement of cross validation accuracy score.

```
[15]: best_parameters = grid_search.best_estimator_.get_params()
      for param_name in sorted(parameters.keys()):
          print(f'{param_name} : {best_parameters[param_name]}')

      clf_max_depth : 5.0
      clf_min_samples_leaf : 1
      clf_min_samples_split : 2

[16]: #Training the model with the optimised parameters
      tree_clf = DecisionTreeClassifier(max_depth=5,min_samples_leaf=1,min_samples_split=2)
      tree_clf.fit(train_f,train_t)

      #KFold cross-validation for evaluating the model
      cv = KFold(n_splits=10, random_state=1, shuffle=True)
      scores = cross_val_score(tree_clf, train_f, train_t, scoring='accuracy', cv=cv, n_jobs=-1)

      # Reporting the performance
      print('Accuracy: %.3f (%.3f)' % (np.mean(scores), np.std(scores)))

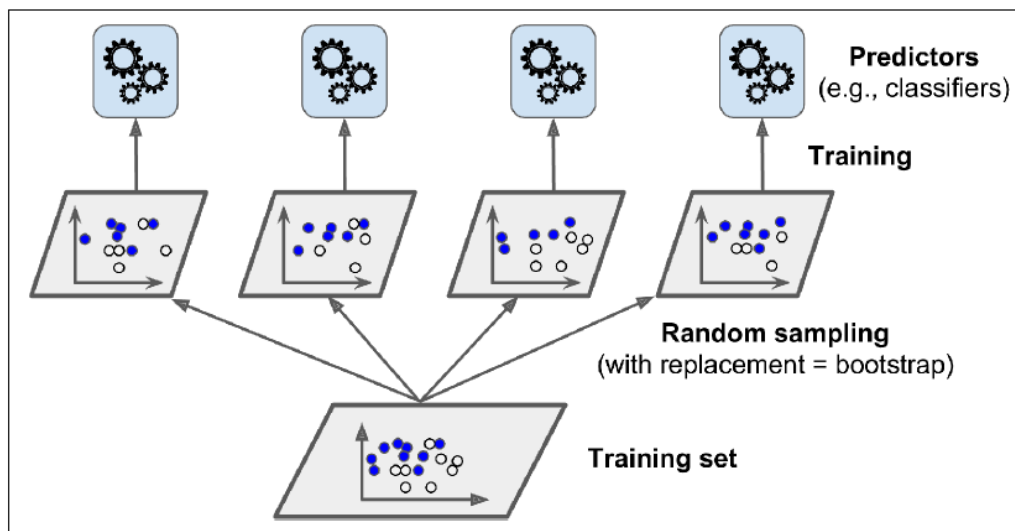
      Accuracy: 0.730 (0.007)
```

Ensemble Methods for Classification:

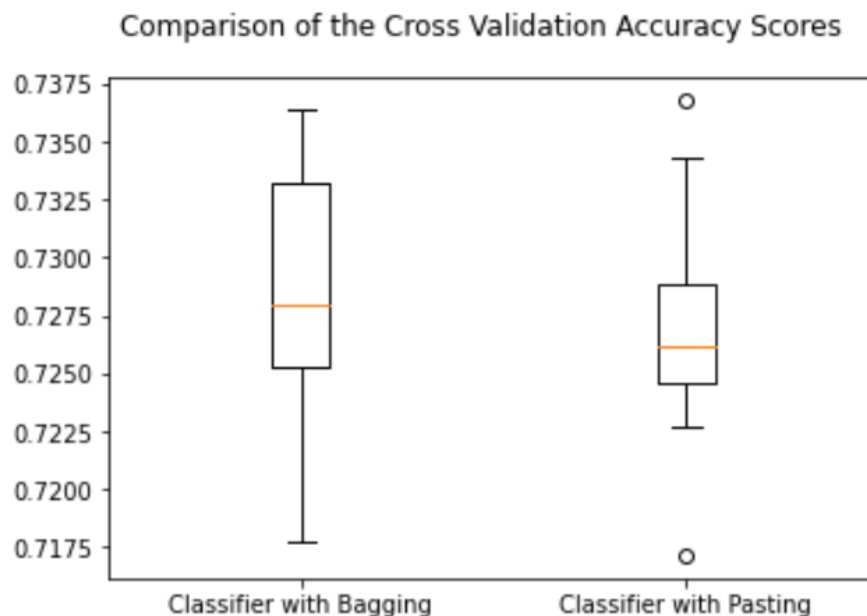
Ensemble learning helps improve machine learning results by combining several models. This approach allows the production of better predictive performance compared to a single model.

Comparison of Bagging and Pasting:

Bagging and pasting involves training several predictors on different random samples of the training set. The difference is in bagging selection of training sets is done with replacement whereas in Pasting it is done without replacement.



The methods were compared by training an ensemble of 500 Decision Tree classifiers and having a minimum sample size of 100, and then the models' cross validation accuracy was compared and plotted as shown below:



Here, ensemble learning with bagging has a higher mean cross validation accuracy as compared to pasting.

Comparison of Hard and Soft Voting:

Predicting the class with the highest class probability, averaged over all the individual classifiers is called soft voting whereas predicting the class probability by taking the statistical mode of the individual classifiers is called hard voting.

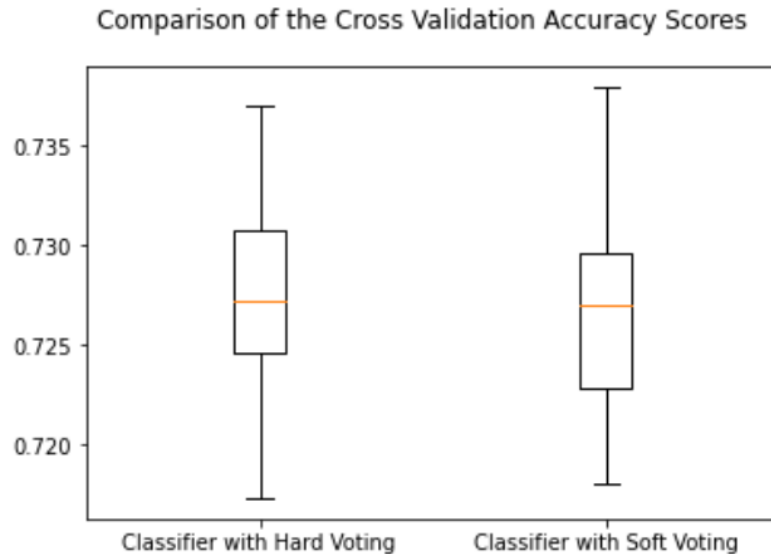
The methods were compared by defining two voting classifier taking in an ensemble of the bagging and pasting classifiers from the previous step

```
hard_voting_clf = VotingClassifier(estimators=[('bagging', bag_clf), ('pasting', pas_clf)], voting='hard')
soft_voting_clf = VotingClassifier(estimators=[('bagging', bag_clf), ('pasting', pas_clf)], voting='soft')

hard_voting_clf.fit(X_train_scaled, y_train)
soft_voting_clf.fit(X_train_scaled, y_train)

plot_crossval_boxplot([('Classifier with Hard Voting', hard_voting_clf),
                       ('Classifier with Soft Voting', soft_voting_clf)])
```

Results obtained show that both hard and soft voting almost make the same predictions and hence their accuracy scores are almost same as given in the plot below.



Random Forest Classifier:

Ensemble method involving multiple decision trees and using bagging to predict the classes of the various instances.

Methodology followed to train the Random Forest Classifier:

The model was created using the default hyper parameters, fit to the training dataset and its accuracy was measured using the K-Fold Cross Validation technique:

```
#Training the model with the default parameters
rf_clf = RandomForestClassifier()
rf_clf.fit(X_train, y_train)

#KFold cross-validation for evaluating the model
cv = KFold(n_splits=10, random_state=1, shuffle=True)
scores = cross_val_score(rf_clf, train_f, train_t, scoring='accuracy', cv=cv, n_jobs=-1)

# Reporting the performance
print('Accuracy: %.3f (%.3f)' % (np.mean(scores), np.std(scores)))
```

Accuracy: 0.712 (0.006)

Following the above step, hyper parameter tuning was performed on the Decision Tree Classifier using the Grid Search Method. The hyperparameters tuned here are:

`min_samples_split` : The minimum number of samples required to split an internal node.

`min_samples_leaf` : The minimum number of samples required to be at a leaf node.

`max_depth` : The maximum depth of the tree.


```

: pipeline = Pipeline([('clf',tree_clf)])
  parameters = {
      'clf__max_depth': np.linspace(1,10,10),
      'clf__min_samples_split': (1,2,3,4,5),
      'clf__min_samples_leaf': (1,2,3,4,5)
  }
  grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1)
  grid_search.fit(train_f,train_t)

  grid_search.best_score_

: 0.7287809523809524

```

The model is then created and trained using the optimized hyperparameters, to result in an improvement of cross validation accuracy score.

```

best_parameters = grid_search.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):
    print(f'{param_name} : {best_parameters[param_name]}')

clf__max_depth : 5.0
clf__min_samples_leaf : 1
clf__min_samples_split : 2

: #Training the model with the optimised parameters
  rf_clf1 = RandomForestClassifier(max_depth=5,min_samples_leaf=1,min_samples_split=2)
  rf_clf1.fit(X_train, y_train)

#KFold cross-validation for evaluating the model
  cv = KFold(n_splits=10, random_state=1, shuffle=True)
  scores = cross_val_score(rf_clf1, train_f, train_t, scoring='accuracy', cv=cv, n_jobs=-1)

# Reporting the performance
  print('Accuracy: %.3f (%.3f)' % (np.mean(scores), np.std(scores)))

Accuracy: 0.729 (0.008)

```

Final Comparison of the different classification algorithms:

	True CVD Absent	False CVD Absent	False CVD Present	True CVD Present	accuracy_score
Logistic_Regression	5505	1550	2142	4540	0.731237
Support_Vector_Machines	5744	1311	2387	4295	0.730800
Random_Forest	5502	1553	2186	4496	0.727815
KNN	5346	1709	2069	4613	0.724976
Naive_Bayes	5488	1567	2968	3714	0.669870
Decision_Tree	4455	2600	2496	4186	0.629031