savinshynu updated the notebook 0a30bb7 · 7 minutes ago History

In this notebook, **we will use logistic regression, decision trees, random forest, Adaboost, gradient boost, xGboost, support vector machine and dense neural network** techniques to solve a classification problem. Then we will fine tune the best model to find the optimized hyperparameters.

For this purpose, we are using the following dataset from kaggle. This dataset consists of heart disease prediction for large population based on their demographic, behavioral and medical risk factors. Every year, 12 million deaths occur across world due to heart diseases and this dataset predicts the probability of someone to have a coronory heart diseases in the next 10 year. Early prognosis and adapting health life styles can aid in avoiding heart diseases.

We will also try to figure out the main leading factors causing coronary heart disease in this population. First we will import the pandas, numpy and matplotlib library for reading and visualization of the data.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Data exploration

In [2]:
```
# reading the dataset
filepath = 'heart_disease.csv' # data file path
df = pd.read_csv(filepath) # read into a pandas dataframe
df.info() # header info
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4238 entries, 0 to 4237
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   male             4238 non-null   int64
 1   age              4238 non-null   int64
```

```
 2   education        4133 non-null   float64
 3   currentSmoker    4238 non-null   int64
 4   cigsPerDay       4209 non-null   float64
 5   BPMeds           4185 non-null   float64
 6   prevalentStroke  4238 non-null   int64
 7   prevalentHyp     4238 non-null   int64
 8   diabetes         4238 non-null   int64
 9   totChol          4188 non-null   float64
 10  sysBP            4238 non-null   float64
 11  diaBP            4238 non-null   float64
 12  BMI              4219 non-null   float64
 13  heartRate        4237 non-null   float64
 14  glucose          3850 non-null   float64
 15  TenYearCHD       4238 non-null   int64
dtypes: float64(9), int64(7)
memory usage: 529.9 KB
```

This dataset consists of 16 features 4238 people participated in this study. This is what each attribute means in the context of this study:

Demographic:

- Sex: male or female(Nominal)
- Age: Age of the patient;(Continuous - Although the recorded ages have been truncated to whole numbers, the concept of age is continuous)
- education : Level of education -- not so relevant feature

Behavioral

- Current Smoker: whether or not the patient is a current smoker (Nominal)
- Cigs Per Day: the number of cigarettes that the person smoked on average in one day. (can be considered continuous as one can have any number of cigarettes, even half a cigarette.) Medical(history)
- BP Meds: whether or not the patient was on blood pressure medication (Nominal)
- Prevalent Stroke: whether or not the patient had previously had a stroke (Nominal)
- Prevalent Hyp: whether or not the patient was hypertensive (Nominal)
- Diabetes: whether or not the patient had diabetes (Nominal)

Medical(current)

- Tot Chol: total cholesterol level (Continuous)
- Sys BP: systolic blood pressure (Continuous)
- Dia BP: diastolic blood pressure (Continuous)
- BMI: Body Mass Index (Continuous)
- Heart Rate: heart rate (Continuous - In medical research, variables such as heart rate though in fact discrete, yet are considered continuous because of large number of possible values.)
- Glucose: glucose level (Continuous)

Predict variable (desired target)

- 10 year risk of coronary heart disease CHD (binary: “1”, means “Yes”, “0” means “No”)

In [3]:
```
df.head(10)  # print the first 10 values
```

Out[3]:

	male	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevale
0	1	39	4.0	0	0.0	0.0	0	
1	0	46	2.0	0	0.0	0.0	0	
2	1	48	1.0	1	20.0	0.0	0	
3	0	61	3.0	1	30.0	0.0	0	
4	0	46	3.0	1	23.0	0.0	0	
5	0	43	2.0	0	0.0	0.0	0	
6	0	63	1.0	0	0.0	0.0	0	
7	0	45	2.0	1	20.0	0.0	0	
8	1	52	1.0	0	0.0	0.0	0	
9	1	43	1.0	1	30.0	0.0	0	

Here the data is already one hot encoded for the all the categorical values.

In [4]:
```
df.describe()
```

Out[4]:

	male	age	education	currentSmoker	cigsPerDay	BPMe
count	4238.000000	4238.000000	4133.000000	4238.000000	4209.000000	4185.000(
mean	0.429212	49.584946	1.978950	0.494101	9.003089	0.0296
std	0.495022	8.572160	1.019791	0.500024	11.920094	0.1695
min	0.000000	32.000000	1.000000	0.000000	0.000000	0.000(
25%	0.000000	42.000000	1.000000	0.000000	0.000000	0.000(
50%	0.000000	49.000000	2.000000	0.000000	0.000000	0.000(
75%	1.000000	56.000000	3.000000	1.000000	20.000000	0.000(
max	1.000000	70.000000	4.000000	1.000000	70.000000	1.000(

In [5]:
```
# let's see if there are any null values
df.isnull().any()
```

Out[5]:
```
male               False
age                False
education           True
currentSmoker      False
cigsPerDay          True
BPMeds              True
prevalentStroke    False
prevalentHyp       False
```

```
diabetes           False
totChol             True
sysBP              False
diaBP              False
BMI                 True
heartRate           True
glucose             True
TenYearCHD         False
dtype: bool
```

There are missing values in multiple columns. We will come back to this later and clean it before modeling. Let's look at the catergorical value statistics first.

In [6]:
```
df["male"].value_counts()
```

Out[6]:
```
male
0    2419
1    1819
Name: count, dtype: int64
```

In [7]:
```
df["currentSmoker"].value_counts()
```

Out[7]:
```
currentSmoker
0    2144
1    2094
Name: count, dtype: int64
```

In [8]:
```
df["BPMeds"].value_counts()
```

Out[8]:
```
BPMeds
0.0    4061
1.0     124
Name: count, dtype: int64
```

In [9]:
```
df["prevalentHyp"].value_counts()
```

Out[9]:
```
prevalentHyp
0    2922
1    1316
Name: count, dtype: int64
```

In [10]:
```
df["prevalentStroke"].value_counts()
```

Out[10]:
```
prevalentStroke
0    4213
1      25
Name: count, dtype: int64
```

In [11]:
```
df["diabetes"].value_counts()
```

Out[11]:
```
diabetes
0    4129
1     109
Name: count, dtype: int64
```

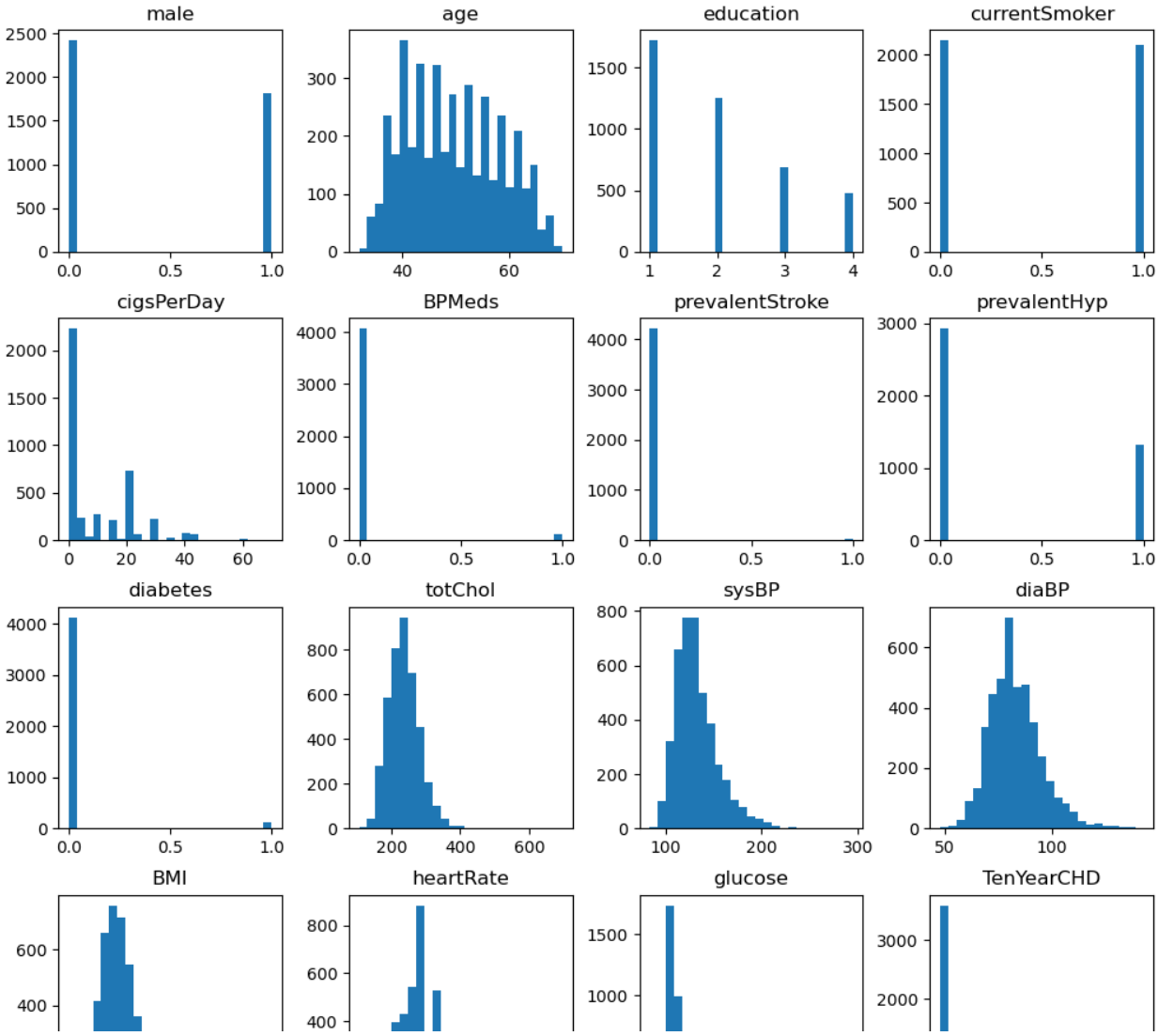
In [12]:
```
df["TenYearCHD"].value_counts()
```

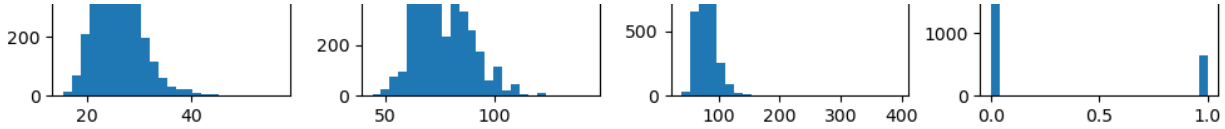
Out[12]:
```
TenYearCHD
0    3594
1     644
Name: count, dtype: int64
```

There are more female in the data compared to male population. There are nearly equal amount of smokers and non-smokers in the dataset. Nearly one third of the population experiences hyper tension, whereas only less than 3 % of the population are on blood pressure meds and have diabetes or had a stroke. Finally, only 15 % of the population has a predicted 10 year risk of coronory heart disease here. The positive class is way smaller than the negative class. ***This is an imabalanced dataset.***

Data Visualization

In [13]:
```
# let's make a histogram plot of different attributes
df.hist(bins = 25, figsize = (12, 12), grid = False)
plt.show()
```





This is a skewed dataset where only ~ 12 % people have risk of cornonoy heart disease. Now let's see if there are any obvious correlations between heard disease chance and the other factors.

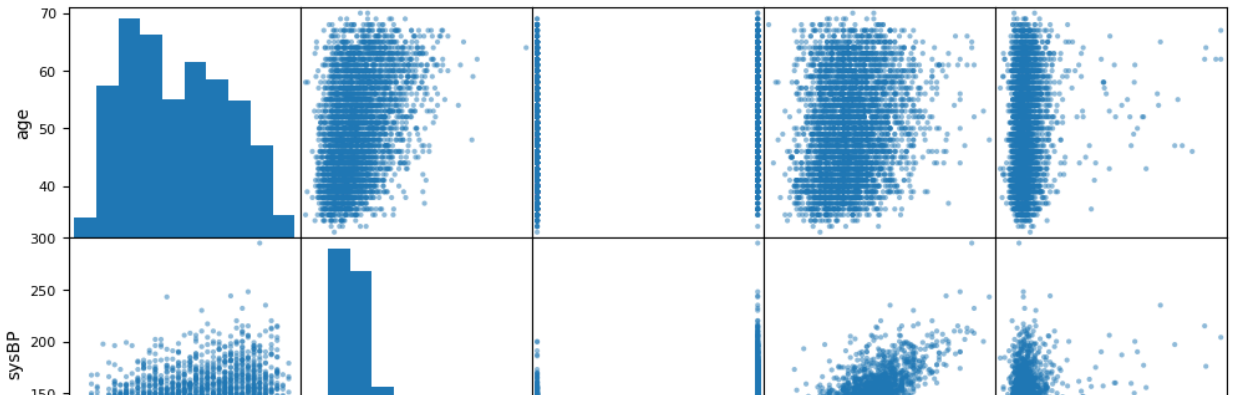
In [14]:
```
# correlation matrix
corr_mat = df.corr()

#correlations corresponding to TenyearCHD
corr_mat["TenYearCHD"].sort_values(ascending = False)
```

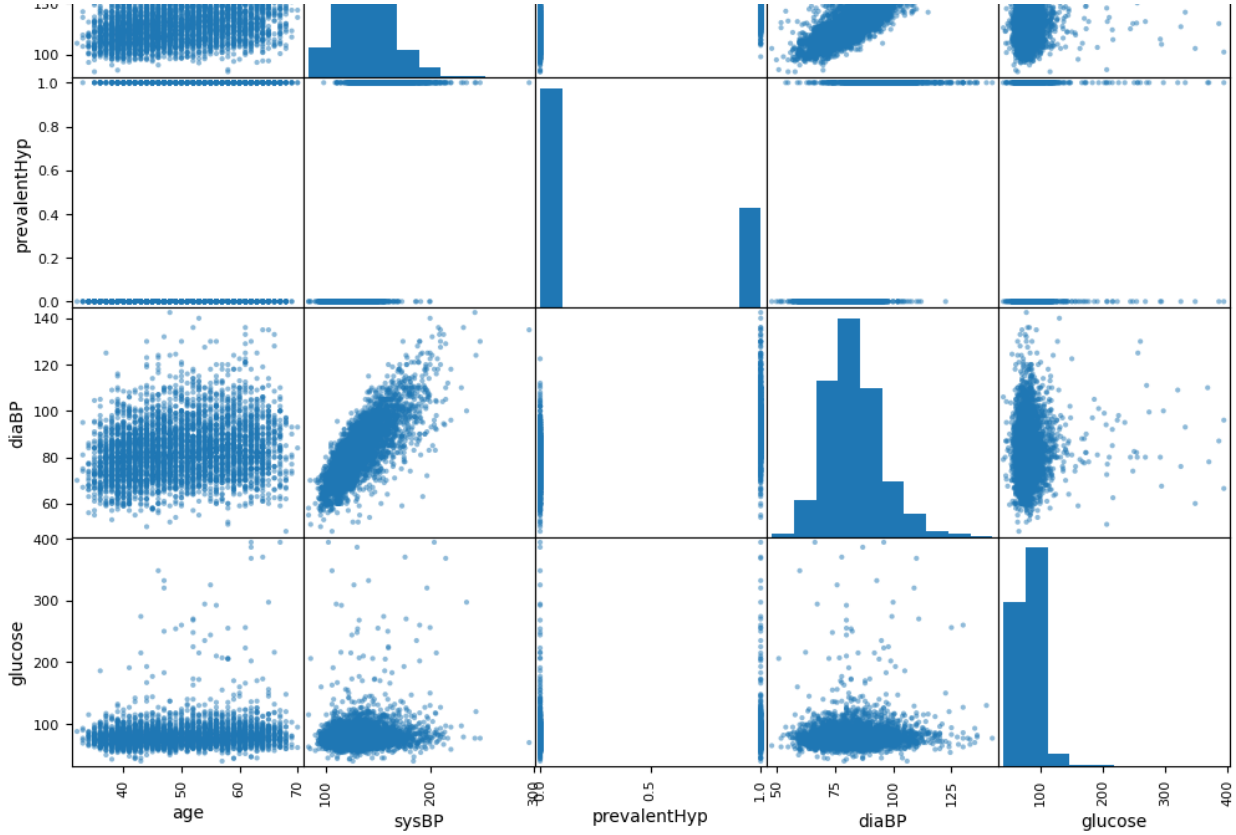
Out[14]:
```
TenYearCHD          1.000000
age                 0.225256
sysBP               0.216429
prevalentHyp        0.177603
diaBP               0.145299
glucose             0.125544
diabetes            0.097317
male                0.088428
BPMeds              0.087489
totChol             0.082184
BMI                 0.075192
prevalentStroke     0.061810
cigsPerDay          0.057884
heartRate           0.022913
currentSmoker       0.019456
education          -0.054059
Name: TenYearCHD, dtype: float64
```

The correlation is kind of interesting here. The 10 year CHD is correlated with age, blood pressure, hypertension and it is not at all corrlated with the total cholestrol, BMI, smoking rate and prevalent stroke.

Let's make scatter matrix of the most correlated features in the dataset.

In [15]:
```
from pandas.plotting import scatter_matrix
attributes = ["age", "sysBP", "prevalentHyp", "diaBP", "glucose"]
scatter_matrix(df[attributes], figsize = (12,12))
plt.show()
```





Here the diabolic BP and systolic BP seems to correlated with each other. Also, age is waekly correlated with the systolic BP. Other than there are not many correlations within the datasets.

Cleaning and Preprocessing the data

After the initial data exploration, now we have some idea of how the dataset looks like. Before doing the classification of the data, we need to do some cleaning and preprocessing.

Since education is not relevant to the heart disease prediction, let's drop that column. Also, we will replace the NaN or Null values with their median values.

In [16]:
```
# drop the education column
df_new = df.drop(columns=["education"])
df_new.head()
```

Out[16]:

	male	age	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevalentHyp	diab
0	1	39	0	0.0	0.0	0	0	
1	0	46	0	0.0	0.0	0	0	
2	1	48	1	20.0	0.0	0	0	
3	0	61	1	30.0	0.0	0	1	
4	0	46	1	23.0	0.0	0	0	

In [17]:
```
# Let's take a look at the Nan values in each columns
df_new.isnull().any()
```

Out[17]:
```
male               False
age                False
currentSmoker      False
cigsPerDay          True
BPMeds              True
prevalentStroke    False
prevalentHyp       False
diabetes           False
totChol             True
sysBP              False
diaBP              False
BMI                 True
heartRate           True
glucose             True
TenYearCHD         False
dtype: bool
```

In [18]:
```
# Here we will look at case by case
#cigsPerDay
print(df_new[np.isnan(df_new["cigsPerDay"])].shape)

# BpMeds
print(df_new[np.isnan(df_new["BPMeds"])].shape)
```

```
(29, 15)
(53, 15)
```

As you can see, at least in 2 columns there are multiple rows with Nan values. We can actaully fill each of these missing rows with their median values. For that we can use df.fillna() function. Since this needs to be carried out for multiple columns, we can use the SimpleImputer from scikit Learn. Imputation is the process of replacing missing values with some value which could be zero, mean or median.

In [19]:
```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy = "median") # replacing nan with median valu
imputer.fit(df_new)

print(imputer.statistics_)
print(df_new.median().values)
```

```
[  0.   49.    0.    0.    0.    0.    0.    0.  234.  128.   82.   25.4
  75.   78.    0. ]
[  0.   49.    0.    0.    0.    0.    0.    0.  234.  128.   82.   25.4
  75.   78.    0. ]
```

In [20]:
```
# applying the imputer values on each columns
df_imp = imputer.transform(df_new)

#look for nan values again.
np.isnan(df_imp).any()
```

Out[20]: False

In [21]:
```
#The dataset does not have any more null values. The output of the imputer fu
#converted back to #the pandas data frame format.
print(df_new.columns)
print(df_new.index)

# writing the array into a data frame
df_imp = pd.DataFrame(df_imp, columns = df_new.columns, index = df_new.index)
df_imp.head()
```

```
Index(['male', 'age', 'currentSmoker', 'cigsPerDay', 'BPMeds',
       'prevalentStroke', 'prevalentHyp', 'diabetes', 'totChol', 'sysBP',
       'diaBP', 'BMI', 'heartRate', 'glucose', 'TenYearCHD'],
      dtype='object')
RangeIndex(start=0, stop=4238, step=1)
```

Out[21]:

	male	age	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevalentHyp	dial
0	1.0	39.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	46.0	0.0	0.0	0.0	0.0	0.0	
2	1.0	48.0	1.0	20.0	0.0	0.0	0.0	
3	0.0	61.0	1.0	30.0	0.0	0.0	1.0	
4	0.0	46.0	1.0	23.0	0.0	0.0	0.0	

Feature scaling and Transformation

Machine algorithms work well when data is feature scaled. Without scaling most models will be biased towards the feature with larger range of values. Feature scaling also increases the faster convergence of gradient descent algorithms which are used to optimize parameters of multiple ML algorithms.

For feature scaling, we can either to Min-max scaling (also known as normalization) where the values will be rescaled between 0 and 1. The scikit learn MinMaxScaler can set this range for example between -1 to 1. Neural networks seems to work better with a zero mean data.

On the other hand, we can use standardization (also known as Z score normalization) where the scaled values will have zero mean and unit standard deviation. Standardization is less affected by outliers compared to normalization in which most of the data points would be crunched towards zero during scaling.

so let's do the standardization of the data. Before that we need split out the X and Y componenents of the data.

In [22]:
```
Y = df_imp["TenYearCHD"]
X = df_imp.drop(columns = ["TenYearCHD"])
X.head()
```

Out[22]:

	male	age	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevalentHyp	dial
0	1.0	39.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	46.0	0.0	0.0	0.0	0.0	0.0	
2	1.0	48.0	1.0	20.0	0.0	0.0	0.0	
3	0.0	61.0	1.0	30.0	0.0	0.0	1.0	
4	0.0	46.0	1.0	23.0	0.0	0.0	0.0	

In [23]:

```
# scaling of the data
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler() # initiating the scaling
X_scaled = std_scaler.fit_transform(X)
print(X_scaled[:2]) # printing the first 2 rows
```

```
[[ 1.1531919  -1.23495068 -0.98827076 -0.75132224 -0.17361158 -0.07703255
  -0.67110093 -0.16247659 -0.9406004  -1.19590711 -1.08262515  0.28737925
   0.34276147 -0.2013593 ]
 [-0.86715836 -0.41825733 -0.98827076 -0.75132224 -0.17361158 -0.07703255
  -0.67110093 -0.16247659  0.30031282 -0.51518725 -0.15898843  0.7197521
   1.59029076 -0.24509896]]
```

Splitting the data

The values in each column are standardized now. All the categorical values are already one-hot encoded and let's move on to the modeling of the data. Before this we need to split the data into train, cross validation and test sets. We will use 60:20:20 ration for the splitting.

In [24]:

```
# module for splitting the data
from sklearn.model_selection import train_test_split

# first split the data into train and test
X_train, X_test, Y_train, Y_test = train_test_split(X_scaled, Y, test_size=0.

# splitting the test into validation and test set
X_val, X_test, Y_val, Y_test = train_test_split(X_test, Y_test, test_size=0.5

print(f"Training set size :{X_train.shape[0]}, Validation set size :{X_val.sh
```

```
Training set size :2542, Validation set size :848, Test set size :848
```

Now the data has been splitted into train, validation and test datasets.

Data modeling

Here we will go through couple of classification models and find out the best model with the highest classification accuracy. Then later we will fine tune the best model to get the accurate results. In this process, we will use 4 models and they are logistic regression,

decision trees, random forest and gradient boosting algorithms.

Logistic regression

For logistic regression, the model is represented as

$$f_{\mathbf{w},b}(x) = g(\mathbf{w} \cdot \mathbf{x} + b)$$

where function g is the sigmoid function. The sigmoid function is defined as:

$$g(z) = \frac{1}{1+e^{-z}}$$

The regularized logistic cost function for logistic regression is of the form

$$J(\mathbf{w},b) = \frac{1}{m}\sum_{i=0}^{m-1}\left[-y^{(i)}\log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=}^{n-}$$

Here we try to optimize the W, b parameters using the gradient descent method which minimizes the cost funtion.

In [25]:
```
# import the logistic regression libraries
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression(random_state = 42)
log_reg.fit(X_train, Y_train)

y_pred_train = log_reg.predict(X_train)
accuracy = len(np.where((Y_train == y_pred_train))[0])/len(Y_train)
print(f"accuracy of training set: {accuracy}")

# the accurace score can also be obtained using the class function
score_train = log_reg.score(X_train, Y_train)
score_cv = log_reg.score(X_val, Y_val)
print(f" Training accuacy: {score_train}, Validation accuracy : {score_cv}")
```

```
accuracy of training set: 0.8524783634933124
 Training accuacy: 0.8524783634933124, Validation accuracy : 0.8679245283018868
```

Feature engineering

Logistic regression is a linear classifier and if the decision boundaries are not linear, then some feature engineering is needed. Let's add some polynomial features and see if that improves the accuracy of the classifier. For this we need to look at the orginal X data before scaling.

In [26]:
```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree = 3, include_bias=False) # using degree 3 po
x_feat = poly.fit_transform(X)
x_feat_scaled = std_scaler.fit_transform(x_feat)

X_train_feat, X_test_feat, Y_train_feat, Y_test_feat = train_test_split(x_fea
```

```
print(X_train_feat.shape)
```

```
(2542, 679)
```

Now instead of 15 columns, there are 664 additional feature engineered columns.

In [27]:

```
# Applying logistic regression on feature engineered dataset
log_reg_feat = LogisticRegression(random_state = 10, max_iter = 1000)
log_reg_feat.fit(X_train_feat, Y_train_feat)
print(f"Accuracy with feature engineering: {log_reg_feat.score(X_train_feat,
```

```
Accuracy with feature engineering: 0.8697875688434303
```

The accuracy score has not changed much after adding the polyomial features probably suggesting that the decision boundary is mostly linear in this case. For rest of the following procedures, we will just stick to the original dataset which does not have any polynomial features as it will take longer training time for the one with polynomial features.

k-fold cross validation

It looks interesting that the validation accuracy is higher than the training accuracy. Usually the available data is partioned into train, validation and test set. However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets. We can use the Scikit-Learns's k-fold cross validation feature to avoid this problem. In this case, we won't need the cross validation set. The data is split into only training and test set. The training data is split into k non-overlapping subsets called folds. Then k-1 subsets will be used for training and a different single fold would used each time for evaluation resulting in k metrics. The performance measure of the k-fold cross validation would be the average of the metric across k folds.

In [28]:

```
from sklearn.model_selection import cross_val_score

#let's split the input data into just training and test set for this purpose
X_train_diff, X_test_diff, Y_train_diff, Y_test_diff = train_test_split(X_sca
print(f"Training set size :{X_train_diff.shape[0]}, Test set size :{X_test_di

#defing the logistic model here
log_reg_diff = LogisticRegression(random_state = 14)

# lets do 5 fold splitting here
acc_score = cross_val_score(log_reg_diff, X_train_diff, Y_train_diff, cv=5, s

print(acc_score)
```

```
Training set size :3390, Test set size :848
[0.85840708 0.85103245 0.85103245 0.85250737 0.8539823 ]
```

Interestingly, the accuracy with just the cross validation is little higher than k-cross cross validation metric. The accuracy values from k-fold method could be a generalized score.

Precision, Recall, F1 score ROC

Since the true class of the output data is a skewed one, it's better to look at the precision, recall, F1 score compared to the accuracy metric. Let's do that. For this task, let's stick to the training set from the (train, validation and test splitting).

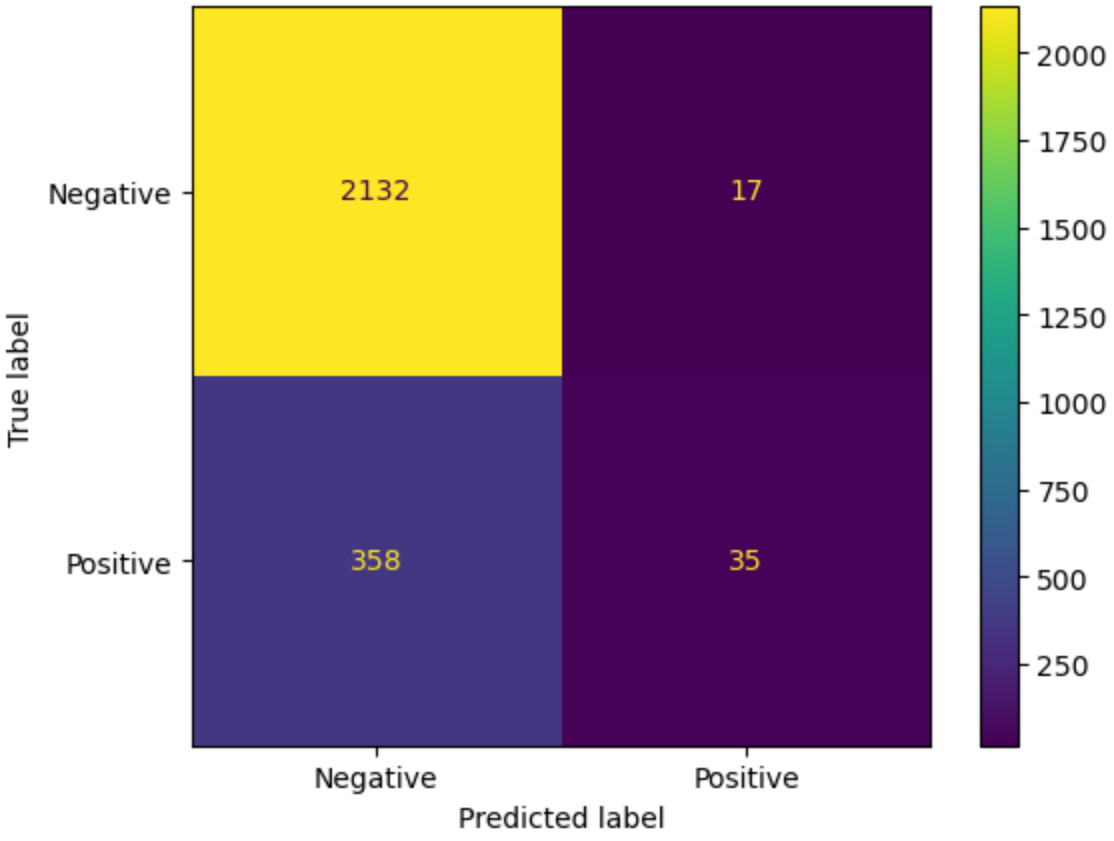
In [29]:
```
# confusion matrix
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(Y_train, y_pred_train)
print(cm)
```

```
[[2132   17]
 [ 358   35]]
```

In the confusion matrix, each row corresponds to the atual class starting from 0. Each column represents the predicted class starting from 0. We can display this is a better way. The data is arranged as

True-Negative	False-Postive
False-Negative	True-Positive

In [30]:
```
from sklearn.metrics import ConfusionMatrixDisplay

ConfusionMatrixDisplay.from_predictions(Y_train, y_pred_train, display_labels
plt.show()
```



Now let's calculate the precision, recall and F1 score on this training set.

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$F1score = \frac{2 \times precision \times Recall}{precision + Recall}$$

In [31]:
```
precision = 35/(35+17)
recall = 35/(35+358)
F1  = 2*precision*recall/(precision + recall)

print(f" precision : {precision}, recall : {recall}, F1 :{F1}")
```

```
 precision : 0.6730769230769231, recall : 0.089058524173028, F1 :0.157303370786
51688
```

In [55]:
```
# We can also do this by sklearn functions
from sklearn.metrics import precision_score, recall_score, f1_score
print(f" Training data: precision : {precision_score(Y_train, y_pred_train)},

#let's calculate the same for the validation dataset as well
ypred_log_val = log_reg.predict(X_val)
ypred_log_test = log_reg.predict(X_test)
print(f" Validation data: precision : {precision_score(Y_val, ypred_log_val)}
```

```
 Training data: precision : 0.6730769230769231, recall : 0.089058524173028, F1
:0.15730337078651688
 Validation data: precision : 0.6, recall : 0.10344827586206896, F1 :0.17647058
823529413
```

Vow, even though the accuracy was high, the classifier has only 67% precision and recall is 8.9 % and F1 score is 0.157 which is way too low. The recall gets little better for the validation data.

In [33]:
```
#Let's look at a detailed classification report
from sklearn.metrics import classification_report
print(classification_report(Y_train, y_pred_train, target_names=['Negative',
```

```
              precision    recall  f1-score   support

    Negative       0.86      0.99      0.92      2149
    Positive       0.67      0.09      0.16       393

    accuracy                           0.85      2542
   macro avg       0.76      0.54      0.54      2542
weighted avg       0.83      0.85      0.80      2542
```

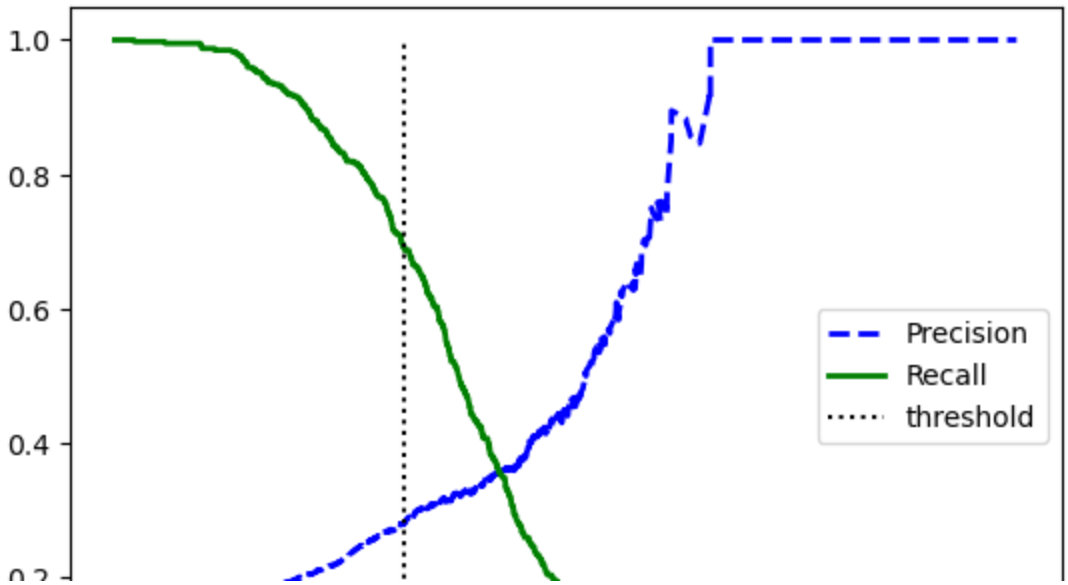
In the next section, we will look at the precision vs recall and ROC curves at different thresholds. There we will see that introducing class weight is equivalent to lowering the threshold which decreases precision and increases the threshold. But the best way to

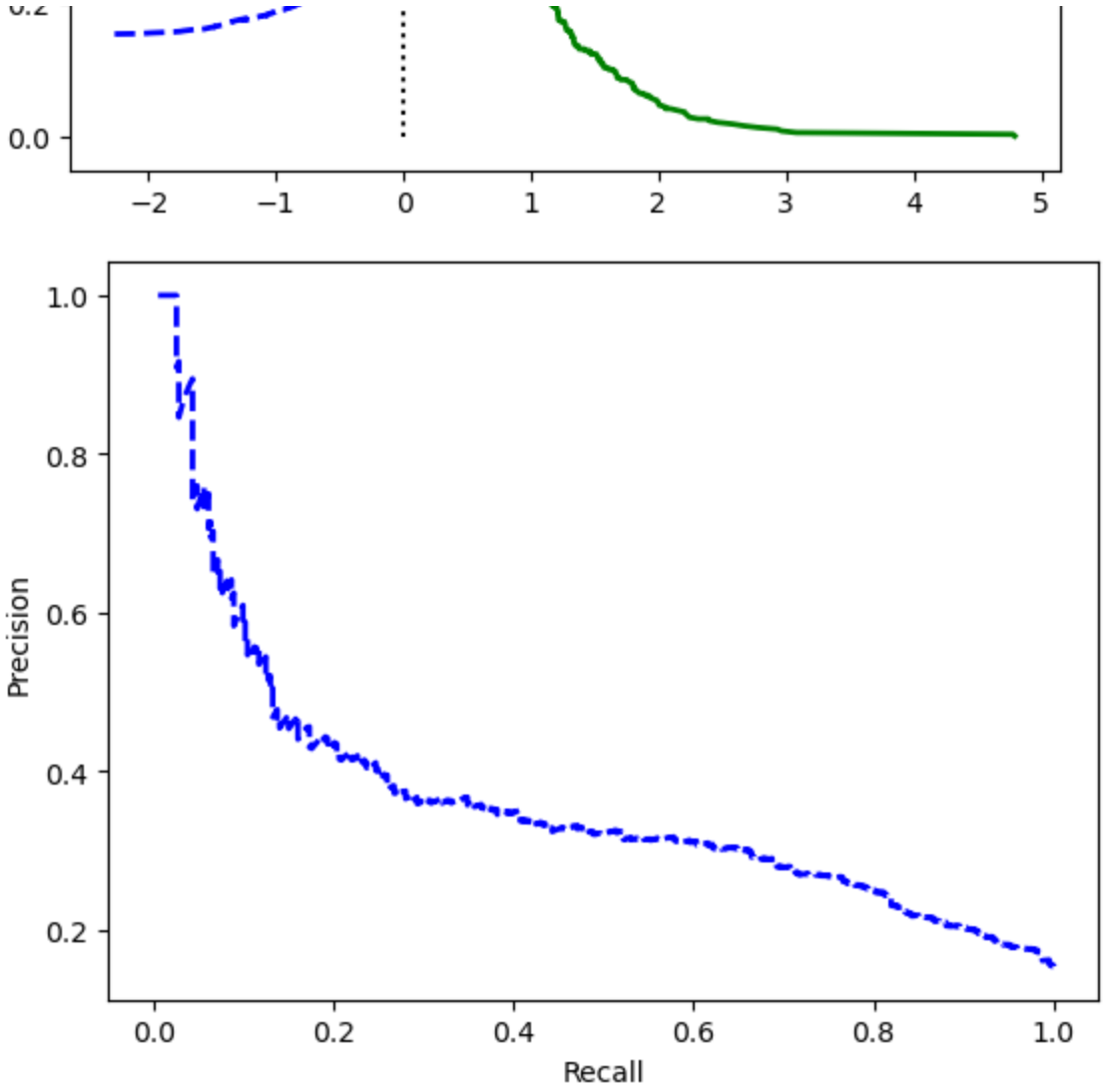
manage imbalanced dataset is getting more data.

In [38]:
```
y_train_prob = log_reg.predict_proba(X_train)
# printing the probability score of 10 examples in each classes.
print(y_train_prob[:10,:])
```

```
[[0.9571339  0.0428661 ]
 [0.7299281  0.2700719 ]
 [0.88042386 0.11957614]
 [0.93728521 0.06271479]
 [0.75307427 0.24692573]
 [0.86918202 0.13081798]
 [0.79374504 0.20625496]
 [0.93503847 0.06496153]
 [0.90650783 0.09349217]
 [0.74798212 0.25201788]]
```

In [39]:
```
# let's do the precision vs recall curve
from sklearn.metrics import precision_recall_curve

#For this first we need the decision score of the classifier.
y_train_scores = log_reg_upd.decision_function(X_train)
#print(y_train_scores)

#calculate the precision and recalls at various thresholds
precisions, recalls, thresholds = precision_recall_curve(Y_train, y_train_sco

#plotting the diagram
plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
plt.vlines(0, 0, 1.0, "k", "dotted", label="threshold")
plt.legend()
plt.show()

#plotting the precision and recall at different thresholds
plt.plot(recalls, precisions, "b--", label="Precision", linewidth=2)
plt.ylabel("Precision")
plt.xlabel("Recall")
plt.show()
```





Here we are dealing with a precision vs Recall (PR) tradeoff situation. As the threshold for classificationg goes up, the precision increaseas and the recall decreases and vice versa. The precision vs recall diagram does not look good and the area under the curve is low. Idealy we want a diagram which bulges out towards the top righ hand corner.

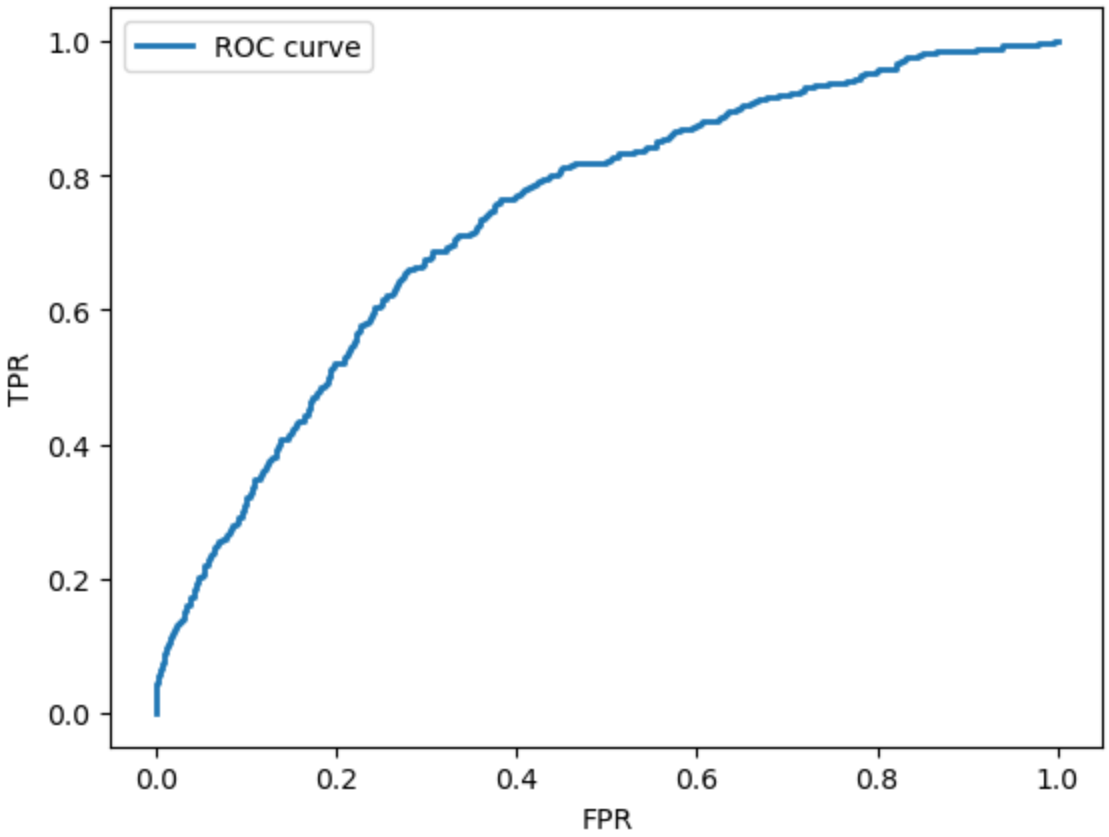
Let's see how the Receiver Operator Characterstics (ROC) curve looks like. In the ROC curve, we plot the true positive rate (tpr) vs the false positive rate (fpr) at different thresholds.

tpr = recall = TP/(TP+FP) = TP/P ,where P is the total positive class.

fpr = FP/(FP+TN) = FP/N, where N is the total negative class.

In [40]:

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(Y_train, y_train_scores)

plt.plot(fpr, tpr, linewidth=2, label="ROC curve")
plt.ylabel("TPR")
plt.xlabel("FPR")
plt.legend()
plt.show()
```



The ROC curve looks pretty good and better compared to the PR curve. **Generally a PR curve is preferred and provide more insights when the true class is skewed in the dataset.** Even though the ROC curve looks better, the PR curve gives better insight into how good the classifier is.

Decision Trees

Decision trees (DTs) are powerful algorithms used for classification and regression purposes. One of the main qualites of decision trees is that they required little data preparation. They don't need feature scaling or standardization. Features can take categorical values. The dataset will be split into multiple leafnodes based on the features. The feature to split the dataset at each node depends on the one which minimizes the Gini impurity or the entropy. Now let's build a DT here.

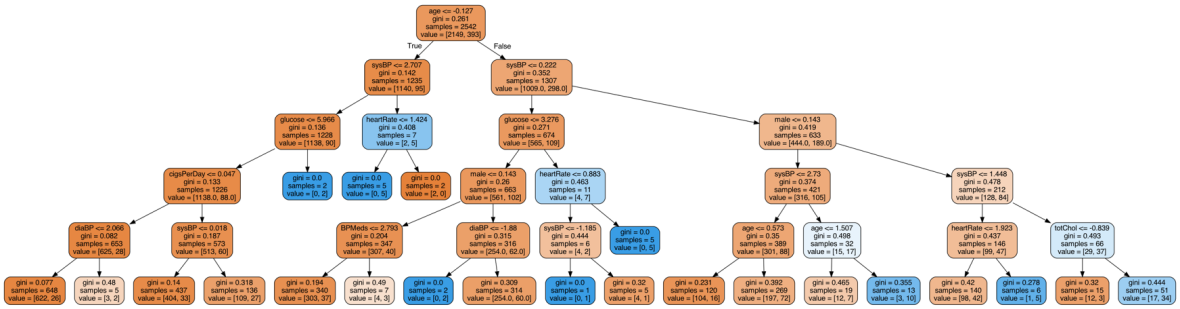
In [41]:
```
# import the DT libraries
from sklearn.tree import DecisionTreeClassifier

#
#let's use the Gini impurity and entropy to see if that makes any differece i
tree_clf1 = DecisionTreeClassifier(criterion = "gini", max_depth=5,  random_s

# let's fit the data
tree_clf1.fit(X_train, Y_train)
```

Out[41]:
```
DecisionTreeClassifier(max_depth=5, random_state=42)
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.

trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [42]:
```
# For tree visualization
from sklearn.tree import export_graphviz
#from graphviz import Source

# Getting a list of the feature names
feat_names = list(X.columns)

# using export_graphviz() for writing down the tree into a dot format
export_graphviz(
            tree_clf1,
out_file="tree_clf1.dot",
feature_names=feat_names,
rounded=True,
filled=True,
impurity=True
)
```

The dot file produced was converted to a .png file which is shown here. Little difficult to see the details since we have a max depth of 5.

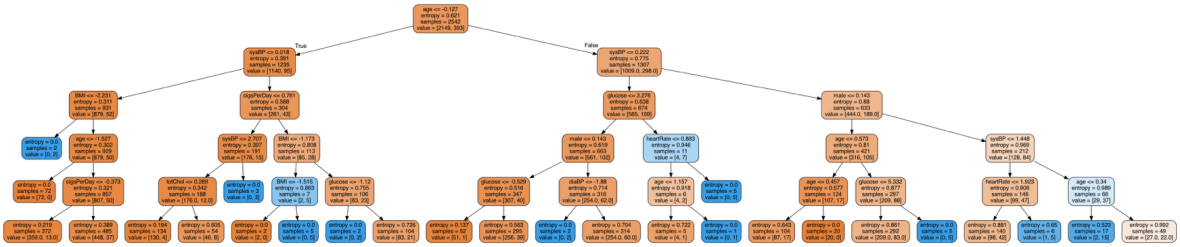


In [43]:
```
# Let's do the same with entropy criterion and see what happens
tree_clf2 = DecisionTreeClassifier(criterion = "entropy", max_depth=5, random
tree_clf2.fit(X_train, Y_train)
```

Out[43]:
```
DecisionTreeClassifier(criterion='entropy', max_depth=5, random_state=
42)
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [44]:
```
export_graphviz(
            tree_clf2,
out_file="tree_clf2.dot",
feature_names=feat_names,
rounded=True,
filled=True,
impurity=True
)
```

Now this is the DT with the entropy criterion:



The ouput DT from the Gini impurity and the entropy criterion are different in this case here. Let's see how their predictions look like. As we can see, there is better intepretability with DTs as we can clearly what the algorithm does in these diagrams.

In [45]:

```
# Get the prediction values
ypred_train_tree1 = tree_clf1.predict(X_train)
ypred_train_tree2 = tree_clf2.predict(X_train)

# Now let's look at the accuracy score from the training and validation data
acc_tree1_train = tree_clf1.score(X_train, Y_train)
acc_tree2_train = tree_clf2.score(X_train, Y_train)

acc_tree1_val = tree_clf1.score(X_val, Y_val)
acc_tree2_val = tree_clf2.score(X_val, Y_val)


print(f" First DT classifier train accuray: {acc_tree1_train}, Second DT clas
print(f" First DT classifier validation accuray: {acc_tree1_val}, Second DT c

print(f"Fist tree: precision : {precision_score(Y_train, ypred_train_tree1)},
print(f"Second tree: precision : {precision_score(Y_train, ypred_train_tree2)
```

```
 First DT classifier train accuray: 0.8623131392604249, Second DT classifier tr
ain accuracy: 0.8619197482297404
 First DT classifier validation accuray: 0.8561320754716981, Second DT classifi
er validation accuracy: 0.8584905660377359
Fist tree: precision : 0.7529411764705882, recall : 0.1628498727735369, F1 :0.2
677824267782427
Second tree: precision : 0.9375, recall : 0.11450381679389313, F1 :0.2040816326
530612
```

The accuracy score of first DT classifier is little higher and the results look similar on the training and validation datasets which is good. However, the accuracy score of DT is only comparable to the logistic regression we used before. But at the same time, precision for the first tree has gone up to 75 % and 93 % for the second tree. The precision for the DT with entropy criterion is higher compared to the DT with the Gini criterion.

Let's increase the tree depth with the entropy criterion and see if we can get an improved metrics with DTs.

In [56]:

```
# Increased tree depth here.
tree_clf3 = DecisionTreeClassifier(criterion = "entropy", max_depth=5, random
tree_clf3.fit(X_train, Y_train)

ypred_train_tree3 = tree_clf3.predict(X_train)
ypred_val_tree3 = tree_clf3.predict(X_val)
```

```
acc_tree3_train = tree_clf3.score(X_train, Y_train)
acc_tree3_val = tree_clf3.score(X_val, Y_val)

print(f" Third DT classifier train accuray: {acc_tree3_train}, Third DT class
print(f" Train precision : {precision_score(Y_train, ypred_train_tree3)}, rec
print(f" Validation precision : {precision_score(Y_val, ypred_val_tree3)}, re
```

```
 Third DT classifier train accuray: 0.8619197482297404, Third DT classifier val
idation accuracy: 0.8584905660377359
 Train precision : 0.9375, recall : 0.11450381679389313, F1 :0.2040816326530612
 Validation precision : 0.25, recall : 0.017241379310344827, F1 :0.032258064516
12903
```

Increasing tree depth increased the training accuracy but the validation accuracy went down which implies that our DT model with higher tree depth is actually overfitting the data. Interstingly increasing the tree depth did not improve the precision, however it improved the recall and thus the F1 score.

Ensemble methods

A single DT is sensitive to the changes in the data. A better approach is to use an ensemble of predictors and aggregate the predictions. Ensemble method works best when the predictors are independent from each other.

Bagging (bootstrap aggregating)

In the case of bagging, we utilize the same training algorithm but we train it on a different random subsets of training set obtained via sampling with replacement. Then the final prediction is determined by the majority of the individual preditions. If the it is trained on random subsets obtained via sampling without replacement, it's called pasting. Now let's try a BaggingClassifier using DT classifier. We also do bagging on other algorithms.

In [50]:
```
from sklearn.metrics import accuracy_score
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

# bagging classifier with DT
bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=300,
                            max_samples=1500, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, Y_train)
ypred_bag_train = bag_clf.predict(X_train)
ypred_bag_val = bag_clf.predict(X_val)

print(f"Bagging DT accuracy score on train:{accuracy_score(Y_train, ypred_bag
print(f" Training data:: precision : {precision_score(Y_train, ypred_bag_trai
print(f" validation data:: precision : {precision_score(Y_val, ypred_bag_val)
```

```
Bagging DT accuracy score on train:0.9736428009441385, validation: 0.8608490566
037735
 Training data:: precision : 1.0, recall : 0.8295165394402035, F1 :0.9068150208
623087
 validation data:: precision : 0.46153846153846156, recall : 0.1034482758620689
6, F1 :0.16901408450704225
```

With just 200 independent estimators, the training accuracy has increased to 97% but the validation accuracy is still only at 86 %. Even though the precision and recall are high for the training data, they are very low for the validation data which is not good. Clearly the model is overfitting the data as we can see 100 % precision for training data.

One of the problems with bagging is that sampling with replacement can result in nearly similary datasets and many trees can end up splitting on the same feature on the root node making it less effective. But still sampling with replacement makes the algorithm robust to small changes in the dataset.

Random Forests

Random Forest (RF) is an ensemble of DTs generally trained via bagging (or pasting). But the algorithm introduces extra randomness by randomizing the feature choice for splitting the node of each DTs. Usually for a DT, the best feature to split each node is determined by selecting the feature split which maximizes the reduction in Gini impurity or the entropy. So instead of searching for the best feature to split the node, RF searches for the best feature within a random subset of features which has a size equivalent to square root of the number of features. This processs happening at each split reduces the correlation between trees and result in lower variance and greater tree diversity. Otherwise, if strong features are present in the data, that will decide the splitting of the trees. Let's see how the algorithm looks like.

In [57]:

```
from sklearn.ensemble import RandomForestClassifier

# define RF class instance
rf_clf = RandomForestClassifier(n_estimators=300, max_samples=1500, n_jobs=-1
rf_clf.fit(X_train, Y_train)

ypred_rf_train = rf_clf.predict(X_train)
ypred_rf_val = rf_clf.predict(X_val)
ypred_rf_test = rf_clf.predict(X_test) # calculating test scores only for RF

print(f" RF accuracy score on train:{accuracy_score(Y_train, ypred_rf_train)}
print(f" Training data:: precision : {precision_score(Y_train, ypred_rf_train
print(f" validation data:: precision : {precision_score(Y_val, ypred_rf_val)}
```

```
 RF accuracy score on train:0.9744295830055075, validation: 0.8679245283018868,
test: 0.8490566037735849
 Training data:: precision : 1.0, recall : 0.8346055979643766, F1 :0.9098474341
192788
 validation data:: precision : 0.6666666666666666, recall : 0.06896551724137931
, F1 :0.125
```

The RF method has a similar accuracy score for train and validation datasets compared to the bagging method. We are not seeing any improvement in the accuracy score on applying the RF to the dataset. However, the precision on the validation data has improved to 66% even though the recall is 7%. The RF model has 85 % accuracy on the test set and 77 % precision on the test set which is interesting.

Boosting

Boosting

Boosting refers to any ensemble method which can combine weak learners into strong learners. Idea of boosting is to train predictors sequentially, where each predictor learns from the mistakes of the previous ones and eventually getting better. The two types are the AdaBoost (adaptive boosting) and gradient boosting.

AdaBoost

In adaptive boosting, each predictor will pay bit more attention to the training examples which were misclassified/mispredicted by the previous tree and those examples will get more weightage during sampling with replacement. This results in predictors focusing more on the difficult training cases. This is a sequential learning process. One drawback is tha you cannot parallelize the boosting process as results of one process is needed for the other. Once the predictors are trained, enseble predictions are similar to bagging, except that predictors have different weights depending on the accuracy on the weighted training set. In the similar way, each training example gets a different weightage, each tree also gets a different amount of say/weightage in the final prediction based on how good of a predictor it was.

Gradient boosting

This method also works sequentially adding predictors to an ensemble correcting its predecessors. Instead of changing the weights of training examples, this method tries to fit a new predictor to the residual errors made by the previous predictor. The final prediction would be the sum of the predictions from multiple trees. The first tree starts with an initial prediction which could be an average representation of the training examples. Next tree would fit on the residuals from the first tree or basically learn from the mistakes of the tree and this process goes on until the residuals get really smaller or we reach end of the maximum number of trees.

Let's try both of these classification mechanisms here.

In [52]:
```
# start with the adaboost here
from sklearn.ensemble import AdaBoostClassifier

# Here we are using a stump here, a DT with depth 1
ada_clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1), n_estimator
ada_clf.fit(X_train, Y_train)

ypred_ada_train = ada_clf.predict(X_train)
ypred_ada_val = ada_clf.predict(X_val)

print(f" Adaboost accuracy score on train:{accuracy_score(Y_train, ypred_ada_
print(f" Training data:: precision : {precision_score(Y_train, ypred_ada_trai
print(f" validation data:: precision : {precision_score(Y_val, ypred_ada_val)
```

```
 Adaboost accuracy score on train:0.8693941778127459, validation: 0.84433962264
15094
 Training data:: precision : 0.7606837606837606, recall : 0.22646310432569974,
F1 :0.3490196078431373
 validation data:: precision : 0.3, recall : 0.10344827586206896, F1 :0.1538461
```

```
validation data:: precision : 0.5, recall : 0.10544627306206896, F1 :0.1358461
5384615385
```

The training and validation metrics are lesser compared to the results from the random forests and the bagging methods. Let's see how gradient boosting performs compared to the adaboosting.

In [53]:

```
# gradient boosting
from sklearn.ensemble import GradientBoostingClassifier

# default is a tree with a max depth of 3
grad_clf = GradientBoostingClassifier(n_estimators=300, learning_rate=0.1, ra
grad_clf.fit(X_train, Y_train)

ypred_grad_train = grad_clf.predict(X_train)
ypred_grad_val = grad_clf.predict(X_val)

print(f" Gradient boost accuracy score on train:{accuracy_score(Y_train, ypre
print(f" Training data:: precision : {precision_score(Y_train, ypred_grad_tra
print(f" validation data:: precision : {precision_score(Y_val, ypred_grad_val
```

```
 Gradient boost accuracy score on train:0.918174665617624, validation: 0.838443
3962264151
 Training data:: precision : 0.9946524064171123, recall : 0.4732824427480916, F
1 :0.6413793103448275
 validation data:: precision : 0.23076923076923078, recall : 0.0775862068965517
3, F1 :0.11612903225806452
```

Comparing the results of the gradient boost wrt to the adaboost, even though the accuracy has increased on the training set, it has gone down in the validation set. Precision has improved a lot on the training set, but it has decreased on the validdation data. Looks like the gradient boosting is overfitting the algorithm.

XGBoost

XGBoost stands for extreme gradient boosting and it is the open source, efficient implementation of gradient boosting. The "extreme" is associated with a ML algorithm with lots of working parts or an extreme versio of the gradient boosting algorithm. This algorithm is similar to the gradient boosting but there are some differences. Instead of fitting a normal DT on the residuals, this algorithm fits an xgboost tree which is little different than the normal DT. In a normal DT, the feature to split the data is decided by the information gain (reduction in entropy or the Gini impurity). However in this case, the feature to split the data is decided by the gain in the similiarity/quality scores while splitting a node. This similiarity score is a metric to calculate how similar are the samples in a leaf are or how well they cluster after splitting a node. The similiarity score when calculated takes into account of the the input regularization parameter. After fitting a xgboost tree to the residuals, the tree undergoes pruning based on a user input tree complexity paramater which reduces overfitting of the data or sensitivity of tree to individual observations.

In [54]:

```
#call the xgboost classifier from the xgboost library
from xgboost import XGBClassifier

#if the
```

```
#If the residuals are not improving, the algorithm will build trees 10 more t
# Here the learning rate decides how much scaling of the contributions from t

xgb_clf =  XGBClassifier(objective='binary:logistic', n_estimators=300, max_d
xgb_clf.fit(X_train, Y_train, verbose=True)

ypred_xgb_train = xgb_clf.predict(X_train)
ypred_xgb_val = xgb_clf.predict(X_val)

print(f" XGboost accuracy score on train:{accuracy_score(Y_train, ypred_xgb_t
print(f" Training data:: precision : {precision_score(Y_train, ypred_xgb_trai
print(f" validation data:: precision : {precision_score(Y_val, ypred_xgb_val)
```

```
 XGboost accuracy score on train:0.8957513768686074, validation: 0.846698113207
5472
 Training data:: precision : 0.9705882352941176, recall : 0.33587786259541985,
F1 :0.49905482041587906
 validation data:: precision : 0.25, recall : 0.0603448275862069, F1 :0.0972222
2222222222
```

The results from the xgboost looks like a regularized version of the gradient boosting. Event then the accuracy and precision are low compared to the results from the random forest method.

Support Vector Machines

Support vector machines are really powerful and versatile models for performing linear and non-linear classification. The model utilizes a hyperplane to maximize the geometric margin (seperation) between different classes. In the case of linearly seperable data, a linear kernel is sufficient. In the case of non-linear data, the base features are transformed to higher dimensions and an optimized hyperplane in the higher dimensions will seperate the different classes. Finding this optimization process involves calculating the dot product between each pair of features. With the help of kernel trick, the dot product between a pair of features in the higher dimensional space is calculated without transforming each coordinate to the higher dimensional space. This allows SVM to operate in low dimensional space and getting the effect of higher dimensional space. We will try to see the effect of linear and a polyonomial kernel in this case.

In [58]:

```
from sklearn.svm import SVC
from sklearn.svm import LinearSVC

#Linear case
svm_lin_clf = LinearSVC(C=1, random_state=42, max_iter=5000) # C is the regul
svm_lin_clf.fit(X_train, Y_train)

ypred_svm_lin_train = svm_lin_clf.predict(X_train)
ypred_svm_lin_val = svm_lin_clf.predict(X_val)

print(f" Linear SVM accuracy score on train:{accuracy_score(Y_train, ypred_sv
print(f" Training data:: precision : {precision_score(Y_train, ypred_svm_lin_
print(f" validation data:: precision : {precision_score(Y_val, ypred_svm_lin_
```

```
 Linear SVM accuracy score on train:0.8505114083398898, validation: 0.865566037
7358491
```

```
 Training data:: precision : 0.8095238095238095, recall : 0.043256997455470736,
F1 :0.08212560386473429
 validation data:: precision : 0.6666666666666666, recall : 0.03448275862068965
5, F1 :0.06557377049180328
```

Even though the metrics are lower for the training data, the model does pretty well on the validation set. Looks like there is higher bias and we need a complex kernel for better classification.

In [61]:

```
#Non-linear case
svm_poly_clf = SVC(kernel="poly", degree=2, C=1, random_state=42) # using a s
svm_poly_clf.fit(X_train, Y_train)

ypred_svm_poly_train = svm_poly_clf.predict(X_train)
ypred_svm_poly_val = svm_poly_clf.predict(X_val)
ypred_svm_poly_test = svm_poly_clf.predict(X_test)

print(f" Polynomial SVM accuracy score on train:{accuracy_score(Y_train, ypre
print(f" Training data:: precision : {precision_score(Y_train, ypred_svm_poly
print(f" validation data:: precision : {precision_score(Y_val, ypred_svm_poly
```

```
 Polynomial SVM accuracy score on train:0.8516915814319433, validation: 0.86556
60377358491, test: 0.8360849056603774
 Training data:: precision : 0.8333333333333334, recall : 0.05089058524173028,
F1 :0.09592326139088728
 validation data:: precision : 0.6666666666666666, recall : 0.03448275862068965
5, F1 :0.06557377049180328
```

Even though training metrics are lesser, the validation scores matches with the metrics from the RF classifier. The precision metrics on test data goes really down for this SVM model Out of all the classifiers, **RF models** seems to be the better performing model on both the validation and test set.

Dense Neural Networks

Using the statistical machine learning techniques, the accuracy have not improved beyond 86% on vaidation data. Also, the recall and F1 score are pretty low. Let's try to see, if a basic neural network model can do a better job in the classification of the data.

For this let's build a simple multilayer perceptron model using tensorflow. Let's build 2 hidden layers, first with 25 units and second with 15 units, and at the end we will have an output layer with sigmoid classification. We will keep the relu activation units for the hidden layers.

In [63]:

```
# load the tensorflow libraries
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Input

# Let's build the model

tf.random.set_seed(42) # Set the random seed to get same reproducable results
mlp_clf = Sequential([
        Input(shape = X_train.shape[1:]),
```

```
        Dense(25, activation="relu", name = 'layer1'),
        Dense(15, activation="relu", name = 'layer2'),
        Dense(5, activation="relu", name = 'layer3'),
        Dense(1, activation="sigmoid", name = 'layer4')
])
mlp_clf.summary()
```

```
2024-08-14 21:48:42.724896: I tensorflow/core/platform/cpu_feature_guard.cc:182
] This TensorFlow binary is optimized to use available CPU instructions in perf
ormance-critical operations.
To enable the following instructions: SSE4.1 SSE4.2, in other operations, rebui
ld TensorFlow with the appropriate compiler flags.
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 layer1 (Dense)              (None, 25)                375

 layer2 (Dense)              (None, 15)                390

 layer3 (Dense)              (None, 5)                 80

 layer4 (Dense)              (None, 1)                 6

=================================================================
Total params: 851
Trainable params: 851
Non-trainable params: 0
_________________________________________________________________
```

In [64]:

```
# Now let's train the model
mlp_clf.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3),
                loss = tf.keras.losses.BinaryCrossentropy(),
                metrics = ["accuracy"])

# Define a custom callback to log the loss after every batch
class BatchLossLogger(tf.keras.callbacks.Callback):
    def on_batch_end(self, batch, logs=None):
        print(f"Batch {batch}: Loss = {logs['loss']}")

#hist = mlp_clf.fit(X_train, Y_train, epochs=2, batch_size=64, validation_dat
hist = mlp_clf.fit(X_train, Y_train, epochs=40, batch_size=64, validation_dat
```

```
Epoch 1/40
40/40 [==============================] - 0s 3ms/step - loss: 0.5458 - accuracy:
0.8127 - val_loss: 0.4361 - val_accuracy: 0.8632
Epoch 2/40
40/40 [==============================] - 0s 1ms/step - loss: 0.4287 - accuracy:
0.8450 - val_loss: 0.3818 - val_accuracy: 0.8656
Epoch 3/40
40/40 [==============================] - 0s 1ms/step - loss: 0.4033 - accuracy:
0.8462 - val_loss: 0.3735 - val_accuracy: 0.8667
Epoch 4/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3941 - accuracy:
0.8474 - val_loss: 0.3712 - val_accuracy: 0.8667
Epoch 5/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3896 - accuracy:
0.8478 - val_loss: 0.3720 - val_accuracy: 0.8656
Epoch 6/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3865 - accuracy:
```

```
0.8482 - val_loss: 0.3692 - val_accuracy: 0.8691
Epoch 7/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3830 - accuracy:
0.8493 - val_loss: 0.3696 - val_accuracy: 0.8691
Epoch 8/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3814 - accuracy:
0.8509 - val_loss: 0.3693 - val_accuracy: 0.8691
Epoch 9/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3797 - accuracy:
0.8509 - val_loss: 0.3691 - val_accuracy: 0.8667
Epoch 10/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3780 - accuracy:
0.8513 - val_loss: 0.3689 - val_accuracy: 0.8679
Epoch 11/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3765 - accuracy:
0.8521 - val_loss: 0.3690 - val_accuracy: 0.8679
Epoch 12/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3752 - accuracy:
0.8501 - val_loss: 0.3695 - val_accuracy: 0.8691
Epoch 13/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3738 - accuracy:
0.8521 - val_loss: 0.3694 - val_accuracy: 0.8667
Epoch 14/40
40/40 [==============================] - 0s 911us/step - loss: 0.3731 - accurac
y: 0.8525 - val_loss: 0.3691 - val_accuracy: 0.8667
Epoch 15/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3716 - accuracy:
0.8537 - val_loss: 0.3692 - val_accuracy: 0.8656
Epoch 16/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3703 - accuracy:
0.8537 - val_loss: 0.3688 - val_accuracy: 0.8679
Epoch 17/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3694 - accuracy:
0.8537 - val_loss: 0.3687 - val_accuracy: 0.8667
Epoch 18/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3685 - accuracy:
0.8548 - val_loss: 0.3690 - val_accuracy: 0.8656
Epoch 19/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3670 - accuracy:
0.8548 - val_loss: 0.3698 - val_accuracy: 0.8644
Epoch 20/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3659 - accuracy:
0.8560 - val_loss: 0.3697 - val_accuracy: 0.8656
Epoch 21/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3647 - accuracy:
0.8552 - val_loss: 0.3693 - val_accuracy: 0.8644
Epoch 22/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3644 - accuracy:
0.8552 - val_loss: 0.3714 - val_accuracy: 0.8644
Epoch 23/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3631 - accuracy:
0.8548 - val_loss: 0.3703 - val_accuracy: 0.8632
Epoch 24/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3620 - accuracy:
0.8572 - val_loss: 0.3697 - val_accuracy: 0.8656
Epoch 25/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3618 - accuracy:
0.8556 - val_loss: 0.3689 - val_accuracy: 0.8644
Epoch 26/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3614 - accuracy:
0.8560 - val_loss: 0.3712 - val_accuracy: 0.8632
```

```
Epoch 27/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3598 - accuracy:
0.8580 - val_loss: 0.3707 - val_accuracy: 0.8620
Epoch 28/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3590 - accuracy:
0.8568 - val_loss: 0.3710 - val_accuracy: 0.8620
Epoch 29/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3580 - accuracy:
0.8580 - val_loss: 0.3724 - val_accuracy: 0.8620
Epoch 30/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3566 - accuracy:
0.8572 - val_loss: 0.3716 - val_accuracy: 0.8597
Epoch 31/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3561 - accuracy:
0.8576 - val_loss: 0.3718 - val_accuracy: 0.8608
Epoch 32/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3553 - accuracy:
0.8588 - val_loss: 0.3726 - val_accuracy: 0.8608
Epoch 33/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3541 - accuracy:
0.8576 - val_loss: 0.3721 - val_accuracy: 0.8597
Epoch 34/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3535 - accuracy:
0.8592 - val_loss: 0.3728 - val_accuracy: 0.8585
Epoch 35/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3531 - accuracy:
0.8592 - val_loss: 0.3741 - val_accuracy: 0.8597
Epoch 36/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3517 - accuracy:
0.8580 - val_loss: 0.3727 - val_accuracy: 0.8585
Epoch 37/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3509 - accuracy:
0.8596 - val_loss: 0.3744 - val_accuracy: 0.8597
Epoch 38/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3499 - accuracy:
0.8607 - val_loss: 0.3742 - val_accuracy: 0.8597
Epoch 39/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3488 - accuracy:
0.8607 - val_loss: 0.3741 - val_accuracy: 0.8608
Epoch 40/40
40/40 [==============================] - 0s 1ms/step - loss: 0.3482 - accuracy:
0.8611 - val_loss: 0.3741 - val_accuracy: 0.8561
```

Adjusting batch size is important. For mini-batch gradient descent, the batch size determines how often the model weights and biases are updated per epoch. Model parameters are updated after each batch (whether it is batch gradient or mini-batch gradient). tensorflow model.fit only reports the averaged loss and accuracy for each epoch in the case of minibatch gradient. Batch size is an important hyperparameter and increasing batch size lowers performance. It'a good idea to keep the batch size smaller, however it increases the noise in loss vs epoch plot.

Increasing batch size drops the learners' ability to generalize. The authors of the paper, “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima”, claim that it is because Large Batch methods tend to result in models that get stuck in local minima. The idea is that smaller batches are more likely to push out local minima and find the Global Minima. More the number of iterations, more likely the the parameters get pushed out of local minima.

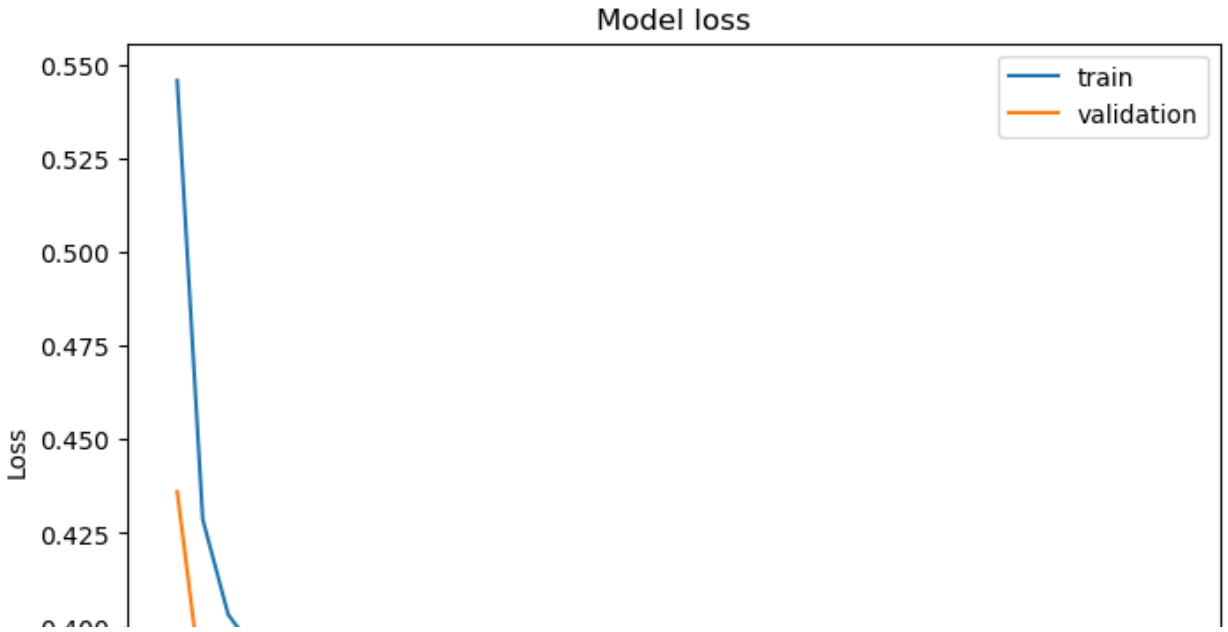
In [65]:

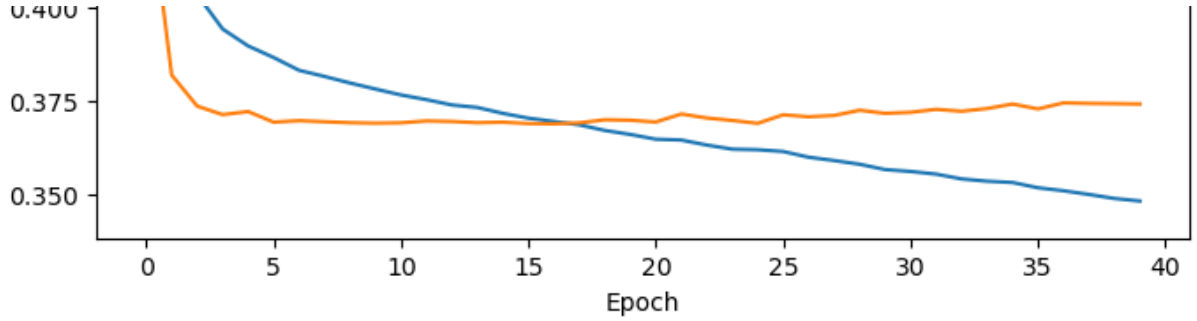
```
# Now let's plot the model loss and accuracy for both the training and valida
# of number of epochs.

# The history.history["loss"] entry is a dictionary with as many values as ep
# model was trained on.
df_loss_acc = pd.DataFrame(hist.history)

# losses data frame
df_loss= df_loss_acc[['loss','val_loss']]
df_loss.rename(columns={'loss':'train','val_loss':'validation'},inplace=True)

print(df_loss.shape)

# accuracy data frame
df_acc= df_loss_acc[['accuracy','val_accuracy']]
df_acc.rename(columns={'accuracy':'train','val_accuracy':'validation'},inplac

# plotting the loss and accuracy
df_loss.plot(title='Model loss',figsize=(8,6)).set(xlabel='Epoch',ylabel='Los
df_acc.plot(title='Model Accuracy',figsize=(8,6)).set(xlabel='Epoch',ylabel='
```

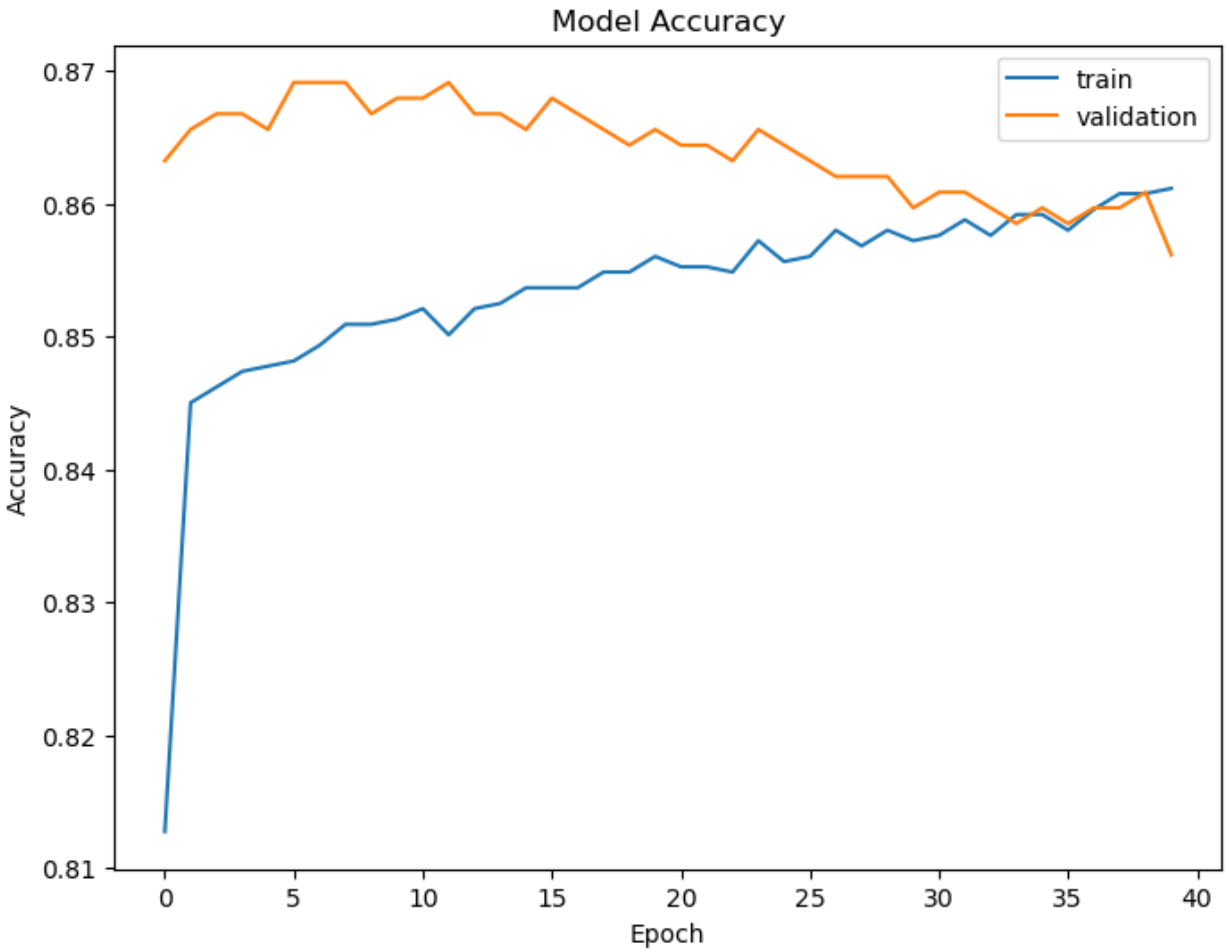
```
(40, 2)
/var/folders/c8/g5hp4hlx7dv6gv7n9zdg74rc0000gn/T/ipykernel_70465/3230200780.py:
10: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/sta
ble/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_loss.rename(columns={'loss':'train','val_loss':'validation'},inplace=True)
/var/folders/c8/g5hp4hlx7dv6gv7n9zdg74rc0000gn/T/ipykernel_70465/3230200780.py:
16: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/sta
ble/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_acc.rename(columns={'accuracy':'train','val_accuracy':'validation'},inplac
e=True)
```

Out[65]:

```
[Text(0.5, 0, 'Epoch'), Text(0, 0.5, 'Accuracy')]
```







The training accuracy is close to 95 %, but unfortunately the validation accuracy is only close to 80%. The performance of neural netowrks is lesser than the tree based searches so far.

In [66]:

```
ypred_mlp_train = mlp_clf.predict(X_train)
ypred_mlp_train[ypred_mlp_train > 0.5] = 1
ypred_mlp_train[ypred_mlp_train < 0.5] = 0

ypred_mlp_val = mlp_clf.predict(X_val)
ypred_mlp_val[ypred_mlp_val > 0.5] = 1
ypred_mlp_val[ypred_mlp_val < 0.5] = 0

print(ypred_mlp_train.shape)
print(f" DNN accuracy score on train:{accuracy_score(Y_train, ypred_mlp_train
print(f" Training data:: precision : {precision_score(Y_train, ypred_mlp_trai
print(f" validation data:: precision : {precision_score(Y_val, ypred_mlp_val)
```

```
80/80 [==============================] - 0s 392us/step
27/27 [==============================] - 0s 383us/step
(2542, 1)
 DNN accuracy score on train:0.8615263571990559, validation: 0.8561320754716981
 Training data:: precision : 0.847457627118644, recall : 0.1272264631043257, F1
:0.22123893805309733
 validation data:: precision : 0.375, recall : 0.07758620689655173, F1 :0.12857
14285714286
```

Let's take a look at the metric scores of the validation data from different models. LR - Logistic regression

DT - Decision Trees

BDT - Bagged DT

RF - Random Forest

AB - Adaboost

GB - Gradient boost

XGB - EXtreme gradient boost

LSVC - Linear support vector classifier

PSVC - Polyonomial kernel SVC

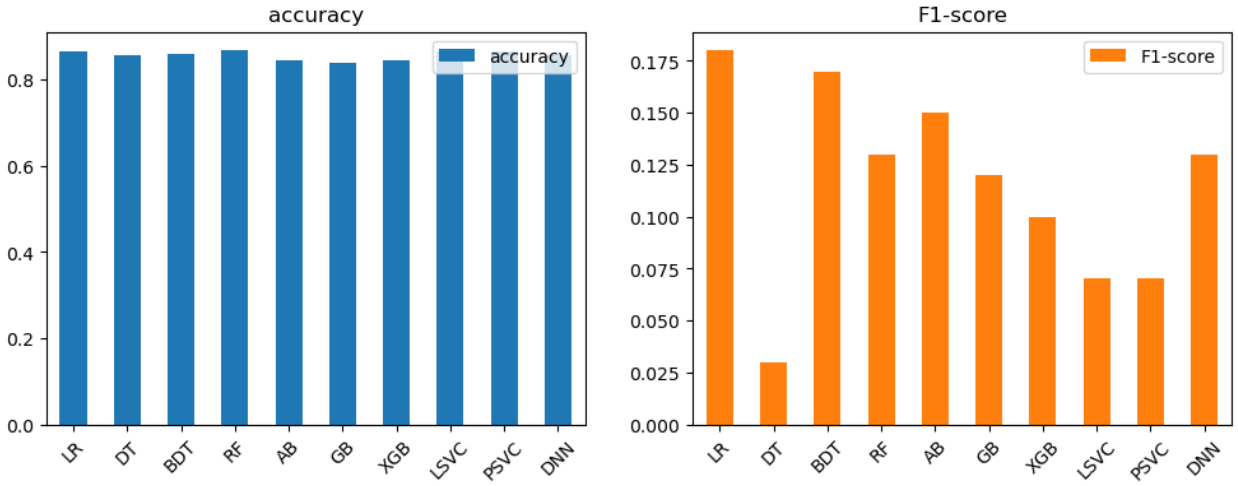
DNN - Dense neural network

Models	Accuracy	Precision	Recall	F1-score
LR	0.867	0.6	0.10	0.18
DT	0.858	0.25	0.02	0.03
BDT	0.861	0.46	0.10	0.17
RF	0.868	0.67	0.07	0.13
AB	0.844	0.3	0.10	0.15
GB	0.838	0.23	0.08	0.12
XGB	0.846	0.25	0.06	0.1
LSVC	0.865	0.67	0.03	0.07
PSVC	0.865	0.67	0.03	0.07
DNN	0.856	0.38	0.08	0.13

In [73]:

```
d = {'accuracy':[0.867, 0.858, 0.861, 0.868, 0.844, 0.838, 0.846, 0.865, 0.86

ind = ['LR', 'DT', 'BDT', 'RF', 'AB', 'GB', 'XGB', 'LSVC', 'PSVC', 'DNN']

res = pd.DataFrame(d, index = ind)
res.plot.bar(rot=45, subplots=True, layout=(1,2), figsize=(12,4))
```

Out[73]:
```
array([[<Axes: title={'center': 'accuracy'}>,
        <Axes: title={'center': 'F1-score'}>]], dtype=object)
```



From the table and the metric plots, all the algorithm have similiar accuracy scores on the validation data. However, RF has the hightest accuracy in this list. The precision varies a lot here and recall is very low for most of the algorithm. However, if I have to choose a model with highest accuracy and F1 score, then **Logistic regression** is the best performing model here. Even though its a simpler algorithm, it performs better classification than the other complicated models.

Imbalanced data issue

If we look at the confusion matrix of the logistic regression model in the beginning of the notebook, most of the positve classes are identified as negative which is a big problem. This happens because the algorithm does not have enough positive examples to learn the features well. This results in a biased model performing poorly on the under represented class. There are 2 solutions:

1. Find more positive sample classes. If not, resample from the existing class. If we just do a sampling with replacement, algorithm will less likely learn anything new. It's basically a duplication.
2. Use a weighted loss function. A weighted loss function is a modification of standard loss function used in training a model. The weights are used to assign a higher penalty to misclassifications of minority class. The idea is to make model more sensitive to minority class by increasing cost of misclassification of that class. The most common way to implement a weighted loss function is to assign higher weight to minority class and lower weight to majority class. The weights can be inversely proportional to frequency of classes, so that minority class gets higher weight and majority class gets lower weight.

Let's try the second approach and see how the results look like. Now let's calculate the class weights first.

weight_for_class_i = total_samples / (num_samples_in_class_i * num_classes)

In [69]:
```
tot_samp = len(Y)
neg_samp = len(Y[Y==0])
pos_samp = len(Y[Y==1])
print(tot_samp, neg_samp, pos_samp)

# calculating class weights
# This value will be multiplied to the corresponding loss calculated for each
weight_0 = tot_samp/(neg_samp*2)
weight_1 = tot_samp/(pos_samp*2)
class_weight = {0:weight_0, 1:weight_1}
print(class_weight)
```

```
4238 3594 644
{0: 0.5895937673900946, 1: 3.290372670807453}
```

In [94]:
```
# Let's try again now with class_weights
log_reg_upd = LogisticRegression(class_weight=class_weight, random_state = 42
log_reg_upd.fit(X_train, Y_train)

y_pred_train_upd = log_reg_upd.predict(X_train)
y_pred_val_upd = log_reg_upd.predict(X_val)
# the accurace score can also be obtained using the class function
score_train = log_reg_upd.score(X_train, Y_train)
score_cv = log_reg_upd.score(X_val, Y_val)
print(f" Training accuacy: {score_train}, Validation accuracy : {score_cv}")
print(f" Training data: precision : {precision_score(Y_train, y_pred_train_up
print(f"  validation data: precision : {precision_score(Y_val, y_pred_val_upd
```

```
 Training accuacy: 0.6805664830841857, Validation accuracy : 0.6709905660377359
 Training data: precision : 0.2824506749740395, recall : 0.6921119592875318, F1
:0.40117994100294985
  validation data: precision : 0.2445141065830721, recall : 0.6724137931034483,
F1 :0.3586206896551724
```

In [95]:
```
ConfusionMatrixDisplay.from_predictions(Y_val, y_pred_val_upd, display_labels
plt.show()
```

In [96]:

```
#Let's look at a detailed classification report
from sklearn.metrics import classification_report
print(classification_report(Y_val, y_pred_val_upd, target_names=['Negative',
```

```
              precision    recall  f1-score   support

    Negative       0.93      0.67      0.78       732
    Positive       0.24      0.67      0.36       116

    accuracy                           0.67       848
   macro avg       0.59      0.67      0.57       848
weighted avg       0.83      0.67      0.72       848
```

The F1 score before introducing class weights are very low for the positive class. In the case with class weights, the F1 scores are reasonbly high enough so that at least one category is not heavily misclassifed.

This also looks like it's tradeoff between the precision and recall. Without the class weights, the precision was high and recall was close to zero. With the class weights, the precision goes down and recall goes up.

Hyperparameter Fine tuning

From this analysis, we have found that the LR model to be the best for predicting the heart diseases. Let's tweak the hyperparameters to get the best prediction out of this. This can be done in 2 ways.

1. Grid search : Basically takes in a user input hyperparameters and evaluate the validation data on all the possible combinations of the models. This method only explore few combinations in the parameter space.
2. Random Search: Rather than using a set of fixed values, this method evaluates a fixed number of combinations, selecting a random values for each hyperparameter at every iteration. This will allow to explore the large parameter space in a better way. This method is good if one hyperparameter does not make any difference and will run only the number of iterations we choose. For eg, if there are 5 hyperparameters with 10 possible values, gridsearch would take 100,000 combinations.

Let's explore both of these cases for the RF search.

In [111…

```
# Grid search
from sklearn.model_selection import GridSearchCV

# Rf instance
#rf = RandomForestClassifier()
```

```
lr = LogisticRegression(class_weight=class_weight, random_state = 42, max_ite

#hyperparemeters
#param_grid = {"n_estimators": [100, 200, 300, 400, 500], "max_samples":[500,

param_grid = {'penalty':['l2'], 'C':[1, 2, 4, 6, 10]}
clf_grid = GridSearchCV(lr, param_grid, scoring='accuracy', cv=3, verbose=1)
clf_grid.fit(X_val, Y_val)
```

```
Fitting 3 folds for each of 5 candidates, totalling 15 fits
```

Out[111…

```
GridSearchCV(cv=3,
             estimator=LogisticRegression(class_weight={0: 0.589593767
3900946,
                                                        1: 3.290372670
807453},
                                          max_iter=200, random_state=4
2),
             param_grid={'C': [1, 2, 4, 6, 10], 'penalty': ['l2']},
             scoring='accuracy', verbose=1)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org

Out[112…

	params	mean_test_score
2	{'C': 4, 'penalty': 'l2'}	0.694518
3	{'C': 6, 'penalty': 'l2'}	0.694518
4	{'C': 10, 'penalty': 'l2'}	0.694518
0	{'C': 1, 'penalty': 'l2'}	0.693336
1	{'C': 2, 'penalty': 'l2'}	0.692153

One thing we can notice is that the different combinations of the hyperparameters are giving better accuracy score than the previous single run on the validation dataset. Now let's try the randomized search.

In [113…

```
# Randomized search
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint, uniform
import numpy.random

#hyperparemeters
#param_grid = {"n_estimators": randint(50,500), "max_samples":randint(200,150

param_grid = {'penalty':['l2'], 'C':randint(1,10)}
clf_rand = RandomizedSearchCV(lr, param_grid, n_iter=20, scoring='accuracy',
clf_rand.fit(X_val, Y_val)
```

```
# converting all the results dictionary into a pandas dataframe for easy view
rand_res = pd.DataFrame(clf_rand.cv_results_)
rand_res[["params", "mean_test_score"]].sort_values(by='mean_test_score', asc
```

```
Fitting 3 folds for each of 20 candidates, totalling 60 fits
```

Out[113…

	params	mean_test_score
0	{'C': 5, 'penalty': 'l2'}	0.694518
1	{'C': 9, 'penalty': 'l2'}	0.694518
18	{'C': 7, 'penalty': 'l2'}	0.694518
17	{'C': 3, 'penalty': 'l2'}	0.694518