csm148finalproj

June 1, 2021

[1]:
```
#Importing the necessary modules
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt # this is used for the plot the graph
import os
import seaborn as sns # used for plot interactive graph.
from sklearn.model_selection import train_test_split, cross_val_score,␣
↪GridSearchCV
from sklearn import metrics
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.cluster import KMeans
from sklearn.metrics import confusion_matrix
import sklearn.metrics.cluster as smc
from sklearn.model_selection import KFold


from matplotlib import pyplot
import itertools

%matplotlib inline

import random

random.seed(42)
```

[2]:
```
def draw_confusion_matrix(y, yhat, classes):
    '''
        Draws a confusion matrix for the given target and predictions
        Adapted from scikit-learn and discussion example.
    '''
    plt.cla()
    plt.clf()
    matrix = confusion_matrix(y, yhat)
    plt.imshow(matrix, interpolation='nearest', cmap=plt.cm.Blues)
```

```
    plt.title("Confusion Matrix")
    plt.colorbar()
    num_classes = len(classes)
    plt.xticks(np.arange(num_classes), classes, rotation=90)
    plt.yticks(np.arange(num_classes), classes)

    fmt = 'd'
    thresh = matrix.max() / 2.
    for i, j in itertools.product(range(matrix.shape[0]), range(matrix.
↪shape[1])):
        plt.text(j, i, format(matrix[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if matrix[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.show()
```

1 1. Loading the Dataset and Basic Stats

[3]:
```
df_main = pd.read_csv('csm148finalprojectdata.csv')
```

[4]:
```
df_main.head()
```

[4]:
```
      id  gender   age  hypertension  heart_disease ever_married  \
0   9046    Male  67.0             0              1          Yes
1  51676  Female  61.0             0              0          Yes
2  31112    Male  80.0             0              1          Yes
3  60182  Female  49.0             0              0          Yes
4   1665  Female  79.0             1              0          Yes

       work_type Residence_type  avg_glucose_level   bmi   smoking_status  \
0        Private          Urban             228.69  36.6  formerly smoked
1  Self-employed          Rural             202.21   NaN     never smoked
2        Private          Rural             105.92  32.5     never smoked
3        Private          Urban             171.23  34.4           smokes
4  Self-employed          Rural             174.12  24.0     never smoked

   stroke
0       1
1       1
2       1
3       1
4       1
```

[5]:
```
#basic stats
df_main.describe()
```

[5]:
```
                 id          age  hypertension  heart_disease  \
count   5110.000000  5110.000000   5110.000000    5110.000000
mean   36517.829354    43.226614      0.097456       0.054012
std    21161.721625    22.612647      0.296607       0.226063
min       67.000000     0.080000      0.000000       0.000000
25%    17741.250000    25.000000      0.000000       0.000000
50%    36932.000000    45.000000      0.000000       0.000000
75%    54682.000000    61.000000      0.000000       0.000000
max    72940.000000    82.000000      1.000000       1.000000

       avg_glucose_level          bmi       stroke
count        5110.000000  4909.000000  5110.000000
mean          106.147677    28.893237     0.048728
std            45.283560     7.854067     0.215320
min            55.120000    10.300000     0.000000
25%            77.245000    23.500000     0.000000
50%            91.885000    28.100000     0.000000
75%           114.090000    33.100000     0.000000
max           271.740000    97.600000     1.000000
```

[6]:
```
df_main.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5110 entries, 0 to 5109
Data columns (total 12 columns):
id                   5110 non-null int64
gender               5110 non-null object
age                  5110 non-null float64
hypertension         5110 non-null int64
heart_disease        5110 non-null int64
ever_married         5110 non-null object
work_type            5110 non-null object
Residence_type       5110 non-null object
avg_glucose_level    5110 non-null float64
bmi                  4909 non-null float64
smoking_status       5110 non-null object
stroke               5110 non-null int64
dtypes: float64(3), int64(4), object(5)
memory usage: 479.1+ KB
```

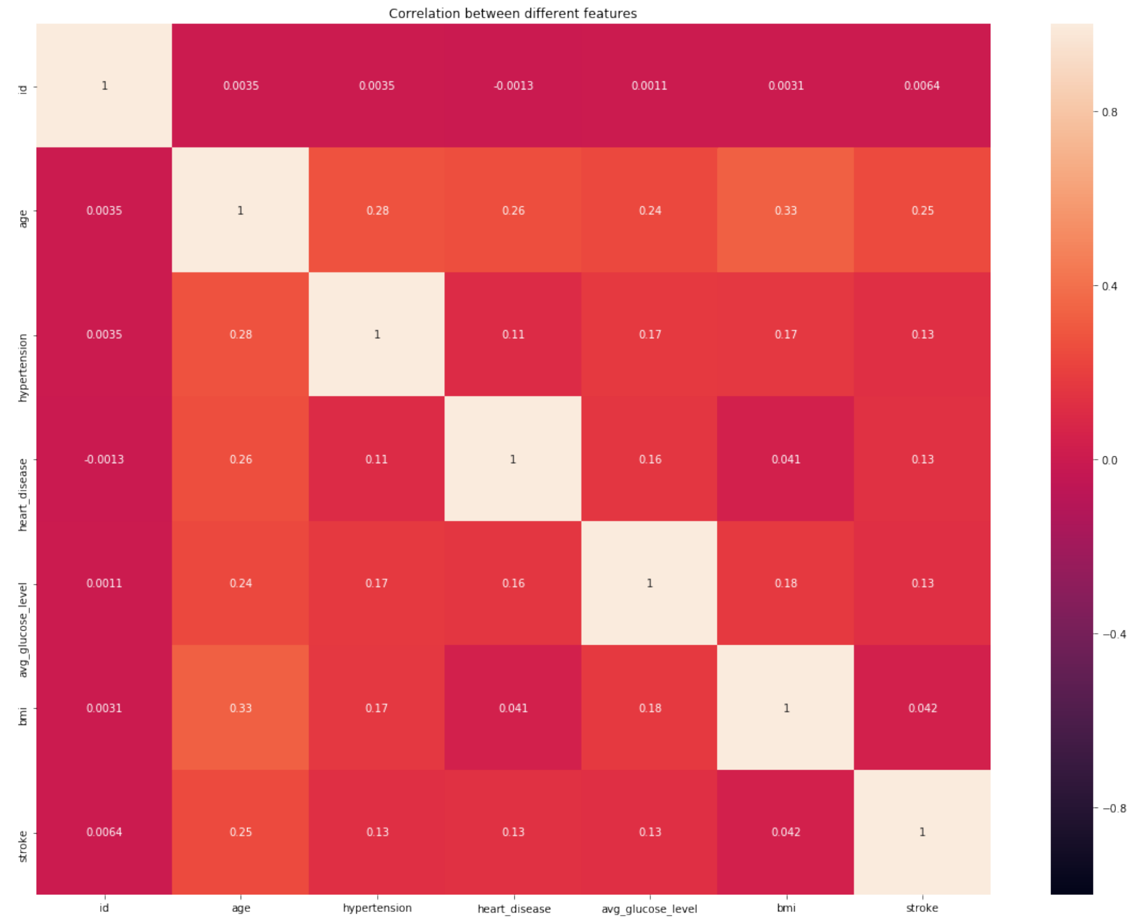
[7]:
```
df_main.shape
```

[7]:
```
(5110, 12)
```

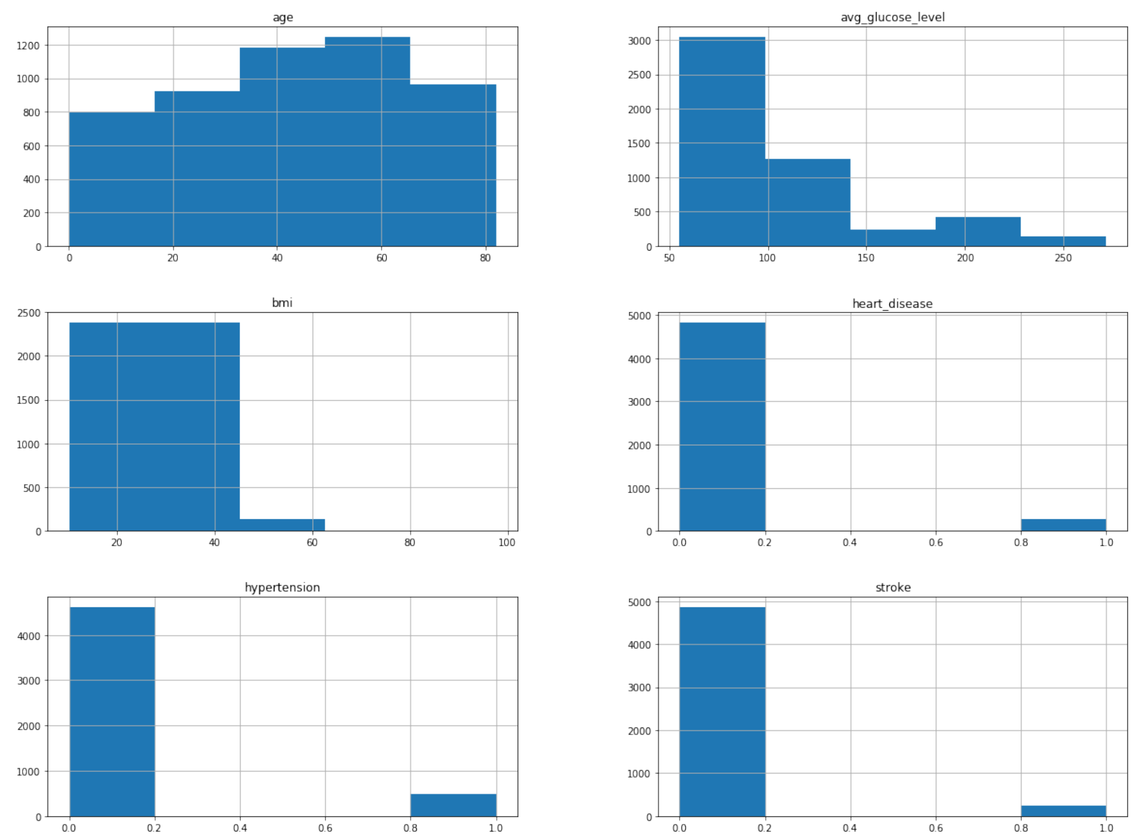
[8]:
```
#Correlation between different labels
plt.figure(figsize=(20, 15))
```

```
hmap = sns.heatmap(df_main.corr(), vmin=-1, vmax=1, annot=True)
hmap.set_title('Correlation between different features')
```

[8]: Text(0.5, 1, 'Correlation between different features')



[9]:
```
df_main = df_main.drop("id", axis=1)
df_main.hist(bins=5, figsize=(20,15))
plt.show()
```



2 2. Pipeline and Data Augmentation

[10]:
```python
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
```

[11]:
```python
df1 = df_main.copy()
```

[12]:
```python
df1.columns
```

[12]:
```
Index(['gender', 'age', 'hypertension', 'heart_disease', 'ever_married',
       'work_type', 'Residence_type', 'avg_glucose_level', 'bmi',
       'smoking_status', 'stroke'],
      dtype='object')
```

2.1 Determining categorization strategy

2.1.1 OHE - one hot encoding

2.1.2 LE - label encoding

[13]:
```
df1['work_type'].unique() #OHE
```

[13]:
```
array(['Private', 'Self-employed', 'Govt_job', 'children', 'Never_worked'],
      dtype=object)
```

[14]:
```
df1['Residence_type'].unique() #LE
```

[14]:
```
array(['Urban', 'Rural'], dtype=object)
```

[15]:
```
df1['smoking_status'].unique() #OHE
```

[15]:
```
array(['formerly smoked', 'never smoked', 'smokes', 'Unknown'],
      dtype=object)
```

[16]:
```
df1['ever_married'].unique() #LE
```

[16]:
```
array(['Yes', 'No'], dtype=object)
```

[17]:
```
df1['gender'].unique() #OHE
```

[17]:
```
array(['Male', 'Female', 'Other'], dtype=object)
```

2.2 Adding a new feature

[18]:
```
#This was done initially to determine what feature to augment and commented out␣
↪on rerunning
#it is done from scratch in the pipeline
#df1['bmi_x_age'] = df1["bmi"]*df1["age"]
```

[19]:
```
#corr_matrix = df1.corr()
#corr_matrix["stroke"].sort_values(ascending=False)
```

[20]:
```
#df1.describe()
```

3 Building the Pipeline

[21]:
```
df1_features = df1.drop("stroke", axis=1)
df1_labels = df1["stroke"].copy()
```

[22]:
```
#Label encoding binary features prior to pipeline transformation
le = LabelEncoder()
for col in ["Residence_type","ever_married"]:
    df1_features[col] = le.fit_transform(df1_features[col])
```

[23]:
```
df1_features.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5110 entries, 0 to 5109
Data columns (total 10 columns):
gender               5110 non-null object
age                  5110 non-null float64
hypertension         5110 non-null int64
heart_disease        5110 non-null int64
ever_married         5110 non-null int64
work_type            5110 non-null object
Residence_type       5110 non-null int64
avg_glucose_level    5110 non-null float64
bmi                  4909 non-null float64
smoking_status       5110 non-null object
dtypes: float64(3), int64(4), object(3)
memory usage: 399.3+ KB
```

[24]:
```
df1_features.head()
```

[24]:
```
   gender   age  hypertension  heart_disease  ever_married      work_type  \
0    Male  67.0             0              1             1        Private
1  Female  61.0             0              0             1  Self-employed
2    Male  80.0             0              1             1        Private
3  Female  49.0             0              0             1        Private
4  Female  79.0             1              0             1  Self-employed

   Residence_type  avg_glucose_level   bmi   smoking_status
0               1             228.69  36.6  formerly smoked
1               0             202.21   NaN     never smoked
2               0             105.92  32.5     never smoked
3               1             171.23  34.4           smokes
4               0             174.12  24.0     never smoked
```

[25]:
```
# This cell implements the complete pipeline for preparing the data
from sklearn.impute import SimpleImputer
from sklearn.base import BaseEstimator, TransformerMixin


imputer = SimpleImputer(strategy="mean") # use mean imputation for missing␣
 ↪values
# remove the categorical features
df1_num = df1_features.drop(["gender", "ever_married", "heart_disease",␣
 ↪"hypertension", "work_type", "Residence_type", "smoking_status"], axis=1)

age_idx, bmi_idx = 0 , 2

class AugmentFeatures(BaseEstimator, TransformerMixin):
    '''
    implements the previous features we had defined
```

```
    df1['bmi_x_age'] = df1["bmi"]*df1["age"]
    '''
    def __init__(self, add_bmi_x_age = True):
        self.add_bmi_x_age = add_bmi_x_age

    def fit(self, X, y=None):
        return self  # nothing else to do

    def transform(self, X):
        final = X
        if self.add_bmi_x_age:
            bmi_x_age = X[:, bmi_idx] * X[:, age_idx]
            final = np.c_[X, bmi_x_age]
        return final

attr_adder = AugmentFeatures()
df1_extra_attribs = attr_adder.transform(df1.values) # generate new features

# this will be are numirical pipeline
# 1. impute, 2. augment the feature set 3. normalize using StandardScaler()
num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="mean")),
        ('attribs_adder', AugmentFeatures()),
        ('std_scaler', StandardScaler()),
    ])

df1_num_tr = num_pipeline.fit_transform(df1_num)

numerical_features = list(df1_num)
categorical_features = ["gender", "work_type", "smoking_status"]

full_pipeline = ColumnTransformer(transformers=[
        ("num", num_pipeline, numerical_features),
        ("cat", OneHotEncoder(), categorical_features)],
    remainder='passthrough')

data_prepared = full_pipeline.fit_transform(df1_features)
```

[26]:
```
data_prepared
```

[26]:
```
array([[ 1.05143428e+00,  2.70637544e+00,  1.00123401e+00, ...,
         1.00000000e+00,  1.00000000e+00,  1.00000000e+00],
       [ 7.86070073e-01,  2.12155854e+00,  4.61555355e-16, ...,
         0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [ 1.62639008e+00, -5.02830130e-03,  4.68577254e-01, ...,
         1.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       ...,
       [-3.63841511e-01, -5.11442636e-01,  2.21736316e-01, ...,
```

```
  0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
[ 3.43796387e-01,  1.32825706e+00, -4.27845098e-01, ...,
  0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
[ 3.42048064e-02, -4.60867458e-01, -3.49895329e-01, ...,
  0.00000000e+00,  1.00000000e+00,  1.00000000e+00]])
```

3.1 Splitting the Dataset

[27]:
```python
Xp_train, Xp_test, Yp_train, Yp_test = train_test_split(data_prepared,␣
 ↪df1_labels, test_size=0.2, random_state=42)
print("Xp_train shape:", Xp_train.shape)
print("Yp_train shape:", Yp_train.shape)
print("Xp_test shape:", Xp_test.shape)
print("Yp_test shape:", Yp_test.shape)
```

```
Xp_train shape: (4088, 20)
Yp_train shape: (4088,)
Xp_test shape: (1022, 20)
Yp_test shape: (1022,)
```

[28]:
```python
Yp_train.value_counts()
```

[28]:
```
0    3901
1     187
Name: stroke, dtype: int64
```

3.2 Balancing the dataset

[29]:
```python
from imblearn.over_sampling import SMOTE
```

[30]:
```python
sm = SMOTE(random_state=20)
Xp_train, Yp_train = sm.fit_resample(Xp_train, Yp_train)
```

[31]:
```python
Yp_train.value_counts()
```

[31]:
```
1    3901
0    3901
Name: stroke, dtype: int64
```

4 3. Logistic Regression

[32]:
```python
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score
```

4.0.1 solver='lbfgs'

[33]:
```
log_clf = LogisticRegression(solver='lbfgs').fit(Xp_train, Yp_train)
y_pred = log_clf.predict(Xp_test)
```

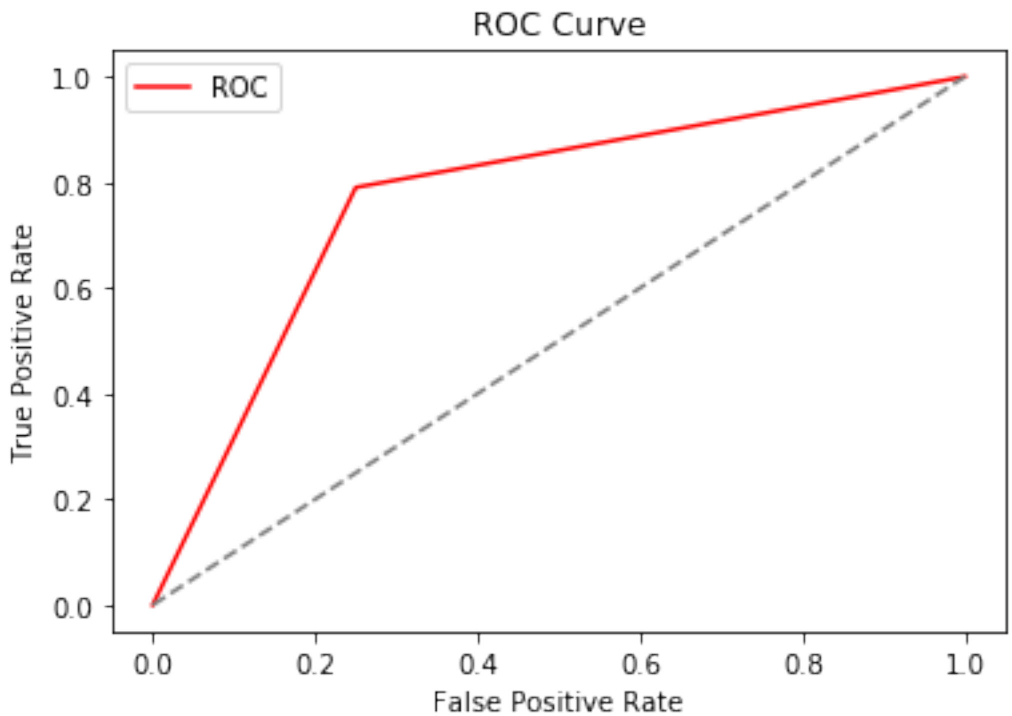
[34]:
```
acc_log = accuracy_score(Yp_test, y_pred)
prec_log = precision_score(Yp_test, y_pred)
rec_log = recall_score(Yp_test, y_pred)
f1_log = f1_score(Yp_test, y_pred)

print("Accuracy:", acc_log)
print("Precision:", prec_log)
print("Recall:", rec_log)
print("F1 Score:", f1_log)
```

```
Accuracy: 0.7524461839530333
Precision: 0.1695501730103806
Recall: 0.7903225806451613
F1 Score: 0.2792022792022792
```

[35]:
```
fpr, tpr, threshold = metrics.roc_curve(Yp_test, y_pred)
plt.plot(fpr, tpr, color='red', label='ROC')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()

auc = np.trapz(tpr,fpr)
print('AUC:', auc)
```




```
AUC: 0.7701612903225806
```

4.0.2 solver='sag'

[36]:
```
log_clf2 = LogisticRegression(penalty='none', solver='sag', max_iter=100).
 ↪fit(Xp_train, Yp_train)
y_pred1 = log_clf2.predict(Xp_test)
```

```
/Users/ojasbardiya/anaconda3/lib/python3.7/site-
packages/sklearn/linear_model/_sag.py:329: ConvergenceWarning: The max_iter was
reached which means the coef_ did not converge
  "the coef_ did not converge", ConvergenceWarning)
```

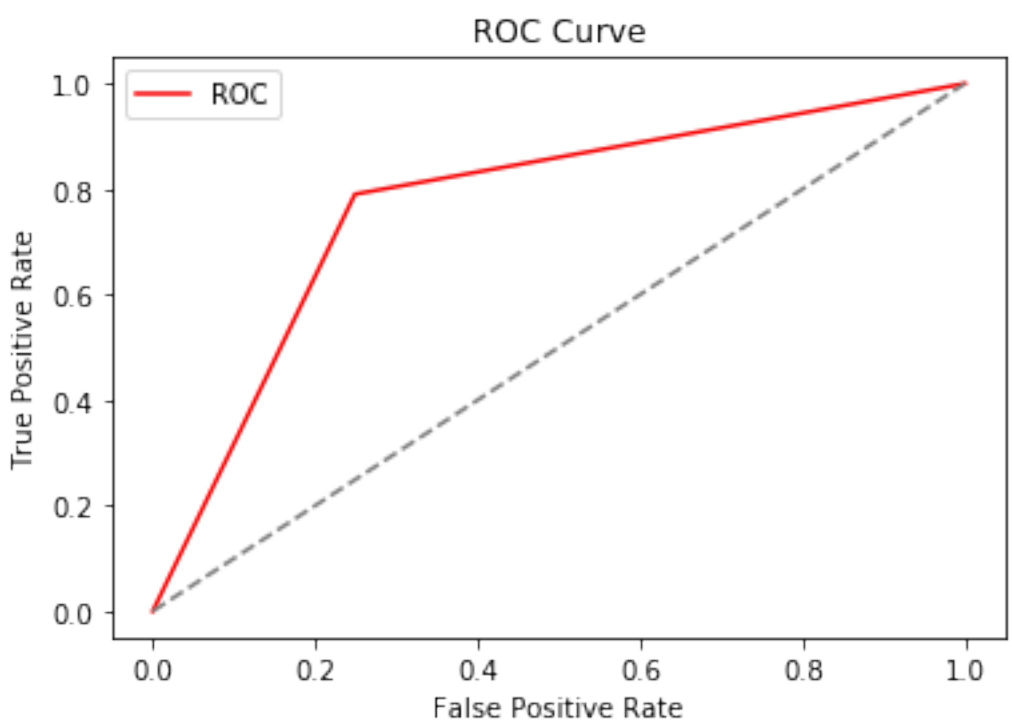
[37]:
```
acc_log = accuracy_score(Yp_test, y_pred1)
prec_log = precision_score(Yp_test, y_pred1)
rec_log = recall_score(Yp_test, y_pred1)
f1_log = f1_score(Yp_test, y_pred1)

print("Accuracy:", acc_log)
print("Precision:", prec_log)
print("Recall:", rec_log)
print("F1 Score:", f1_log)
```

```
Accuracy: 0.7534246575342466
Precision: 0.1701388888888889
Recall: 0.7903225806451613
F1 Score: 0.27999999999999997
```

[38]:
```
fpr, tpr, threshold = metrics.roc_curve(Yp_test, y_pred1)
plt.plot(fpr, tpr, color='red', label='ROC')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()

auc = np.trapz(tpr,fpr)
print('AUC:', auc)
```




```
AUC: 0.7706821236559139
```

4.0.3 Associated p-values and regression

[39]:
```
import statsmodels.api as sms
```

[40]:
```python
# build the OLS model (ordinary least squares) from the training data
stroke_stats = sms.OLS(df1_labels, data_prepared)

# do the fit and save regression info (parameters, etc) in results_stats
results_stats = stroke_stats.fit()
```

[41]:
```python
print(results_stats.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 stroke   R-squared:                       0.085
Model:                            OLS   Adj. R-squared:                  0.082
Method:                 Least Squares   F-statistic:                     27.73
Date:                Tue, 01 Jun 2021   Prob (F-statistic):           3.07e-85
Time:                        12:34:47   Log-Likelihood:                 822.99
No. Observations:                5110   AIC:                            -1610.
Df Residuals:                    5092   BIC:                            -1492.
Df Model:                          17
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
x1             0.0857      0.015      5.696      0.000       0.056       0.115
x2             0.0141      0.003      4.593      0.000       0.008       0.020
x3             0.0030      0.008      0.379      0.705      -0.012       0.018
x4            -0.0187      0.017     -1.107      0.268      -0.052       0.014
x5             0.0352      0.040      0.880      0.379      -0.043       0.114
x6             0.0337      0.040      0.840      0.401      -0.045       0.112
x7             0.0092      0.167      0.055      0.956      -0.318       0.337
x8            -0.0078      0.020     -0.384      0.701      -0.048       0.032
x9             0.0305      0.042      0.731      0.465      -0.051       0.112
x10            0.0067      0.019      0.347      0.728      -0.031       0.044
x11           -0.0127      0.020     -0.626      0.532      -0.052       0.027
x12            0.0614      0.022      2.796      0.005       0.018       0.104
x13            0.0215      0.023      0.939      0.348      -0.023       0.066
x14            0.0227      0.023      0.992      0.321      -0.022       0.068
x15            0.0138      0.023      0.609      0.542      -0.031       0.058
x16            0.0200      0.023      0.861      0.389      -0.026       0.066
x17            0.0389      0.010      3.776      0.000       0.019       0.059
x18            0.0501      0.013      3.714      0.000       0.024       0.077
x19           -0.0348      0.009     -4.061      0.000      -0.052      -0.018
x20            0.0052      0.006      0.907      0.365      -0.006       0.017
==============================================================================
Omnibus:                     3802.843   Durbin-Watson:                   0.172
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            47486.467
Skew:                           3.646   Prob(JB):                         0.00
Kurtosis:                      16.032   Cond. No.                     1.88e+16
==============================================================================
```

```
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The smallest eigenvalue is 3.92e-29. This might indicate that there are
strong multicollinearity problems or that the design matrix is singular.
```

5 4. PCA

[42]:
```python
from sklearn import decomposition
```

[43]:
```python
pca_data = data_prepared.copy()
```

[44]:
```python
pca_data
```

[44]:
```
array([[ 1.05143428e+00,  2.70637544e+00,  1.00123401e+00, ...,
         1.00000000e+00,  1.00000000e+00,  1.00000000e+00],
       [ 7.86070073e-01,  2.12155854e+00,  4.61555355e-16, ...,
         0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [ 1.62639008e+00, -5.02830130e-03,  4.68577254e-01, ...,
         1.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       ...,
       [-3.63841511e-01, -5.11442636e-01,  2.21736316e-01, ...,
         0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [ 3.43796387e-01,  1.32825706e+00, -4.27845098e-01, ...,
         0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [ 3.42048064e-02, -4.60867458e-01, -3.49895329e-01, ...,
         0.00000000e+00,  1.00000000e+00,  1.00000000e+00]])
```

[45]:
```python
pca = decomposition.PCA(0.8)

# Now we run the fit operation to convert our
# data to a PCA transformmed data
pca_data = pca.fit_transform(pca_data)
```

[46]:
```python
pca_data.shape
```

[46]:
```
(5110, 6)
```

5.1 Splitting the data

[47]:
```python
new_X_train, new_X_test, new_Y_train, new_Y_test = train_test_split(pca_data,␣
↪df1_labels, test_size=0.2, random_state=42)
```

5.2 Balancing the Data

[48]:
```
sm = SMOTE(random_state=20)
new_X_train, new_Y_train = sm.fit_resample(new_X_train, new_Y_train)
```

[49]:
```
new_Y_train.value_counts()
```

[49]:
```
1    3901
0    3901
Name: stroke, dtype: int64
```

5.3 Implementing Logistic Regression after PCA transformation

5.4 solver='lbgfs'

[50]:
```
log_clf = LogisticRegression(solver='lbfgs').fit(new_X_train, new_Y_train)
y_pred = log_clf.predict(new_X_test)
```

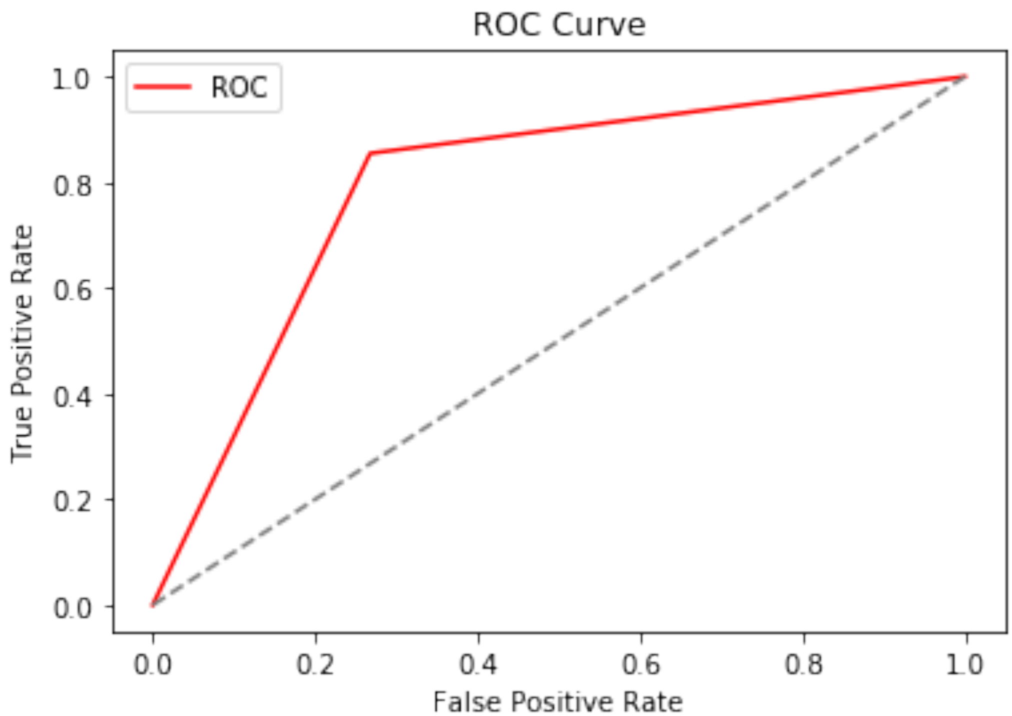
[51]:
```
acc_log = accuracy_score(new_Y_test, y_pred)
prec_log = precision_score(new_Y_test, y_pred)
rec_log = recall_score(new_Y_test, y_pred)
f1_log = f1_score(new_Y_test, y_pred)

print("Accuracy:", acc_log)
print("Precision:", prec_log)
print("Recall:", rec_log)
print("F1 Score:", f1_log)
```

```
Accuracy: 0.7397260273972602
Precision: 0.17096774193548386
Recall: 0.8548387096774194
F1 Score: 0.2849462365591398
```

[52]:
```
fpr, tpr, threshold = metrics.roc_curve(Yp_test, y_pred)
plt.plot(fpr, tpr, color='red', label='ROC')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()

auc = np.trapz(tpr,fpr)
print('AUC:', auc)
```




```
AUC: 0.793565188172043
```

5.5 solver='sag'

[53]:
```
log_clf2 = LogisticRegression(penalty='none', solver='sag', max_iter=100).
↪fit(new_X_train, new_Y_train)
y_pred1 = log_clf2.predict(new_X_test)
```

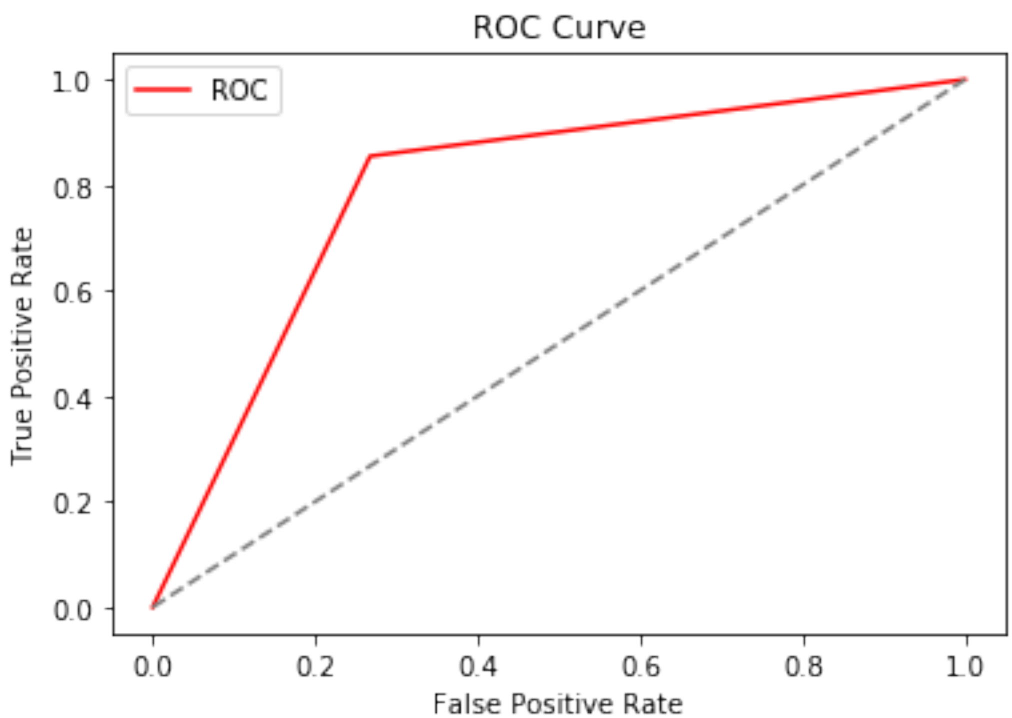
[54]:
```
acc_log = accuracy_score(new_Y_test, y_pred1)
prec_log = precision_score(new_Y_test, y_pred1)
rec_log = recall_score(new_Y_test, y_pred1)
f1_log = f1_score(new_Y_test, y_pred1)

print("Accuracy:", acc_log)
print("Precision:", prec_log)
print("Recall:", rec_log)
print("F1 Score:", f1_log)
```

```
Accuracy: 0.7397260273972602
Precision: 0.17096774193548386
Recall: 0.8548387096774194
F1 Score: 0.2849462365591398
```

[55]:
```
fpr, tpr, threshold = metrics.roc_curve(Yp_test, y_pred1)
plt.plot(fpr, tpr, color='red', label='ROC')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()

auc = np.trapz(tpr,fpr)
print('AUC:', auc)
```




```
AUC: 0.793565188172043
```

6 5. Bagging

[56]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn import model_selection
```

[57]:
```
tree = DecisionTreeClassifier(max_depth=3, random_state=20)
```

6.1 Using non-PCA data

[58]:
```
bagging = BaggingClassifier(base_estimator=tree, n_estimators=16,␣
 ↪max_samples=200, bootstrap=True)
```

[59]:
```
bagging.fit(Xp_train, Yp_train)
```

[59]:
```
BaggingClassifier(base_estimator=DecisionTreeClassifier(max_depth=3,
                                                        random_state=20),
                  max_samples=200, n_estimators=16)
```

[60]:
```
y_pred = bagging.predict(Xp_test)
```

[61]:
```
acc_bag = accuracy_score(Yp_test, y_pred)
prec_bag = precision_score(Yp_test, y_pred)
rec_bag = recall_score(Yp_test, y_pred)
f1_bag = f1_score(Yp_test, y_pred)

print("Accuracy:", acc_bag)
print("Precision:", prec_bag)
print("Recall:", rec_bag)
print("F1 Score:", f1_bag)
```

```
Accuracy: 0.7397260273972602
Precision: 0.16883116883116883
Recall: 0.8387096774193549
F1 Score: 0.28108108108108104
```

6.2 Using PCA data

[62]:
```
bagging2 = BaggingClassifier(base_estimator=tree, n_estimators=6,␣
 ↪max_samples=200, bootstrap=True)
```

[63]:
```
bagging2.fit(new_X_train, new_Y_train)
```

[63]:
```
BaggingClassifier(base_estimator=DecisionTreeClassifier(max_depth=3,
                                                        random_state=20),
                  max_samples=200, n_estimators=6)
```

[64]:
```
y_pred = bagging2.predict(new_X_test)
```

[65]:
```
acc_bag = accuracy_score(new_Y_test, y_pred)
prec_bag = precision_score(new_Y_test, y_pred)
rec_bag = recall_score(new_Y_test, y_pred)
f1_bag = f1_score(new_Y_test, y_pred)

print("Accuracy:", acc_bag)
print("Precision:", prec_bag)
print("Recall:", rec_bag)
print("F1 Score:", f1_bag)
```

```
Accuracy: 0.7005870841487280
Precision: 0.1534090909090909
Recall: 0.8709677419354839
F1 Score: 0.2608695652173913
```

6.3 Hyperparameter tuning (using only PCA transformed data)

[77]:
```python
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.metrics import f1_score, make_scorer
```

[78]:
```python
f1 = make_scorer(f1_score , average='macro')
```

[79]:
```python
n_estimators = [10, 50, 100]
max_samples = [0.6, 0.8, 1.0]
max_features = [4, 5, 6]
```

[80]:
```python
grid = dict(n_estimators = n_estimators, max_samples = max_samples,
              max_features = max_features)
grid_search =␣
 ↪GridSearchCV(BaggingClassifier(base_estimator=DecisionTreeClassifier()),␣
 ↪param_grid=grid, n_jobs=-1, cv=5, scoring=f1)
```

[81]:
```python
grid_result = grid_search.fit(new_X_train, new_Y_train)
```

[82]:
```python
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
Best: 0.936358 using {'max_features': 4, 'max_samples': 1.0, 'n_estimators':
100}
```

7 6. Neural Network

Note: From this point on, only PCA tranformed data is used.

[72]:
```python
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_classification
```

[73]:
```python
#We only use PCA transformed data from this point on
mlp_clf = MLPClassifier(hidden_layer_sizes=(150,100,50), max_iter=200, alpha=0.
 ↪0001,
                        solver='adam', verbose=10,  random_state=21)
mlp_clf.fit(new_X_train, new_Y_train)

y_pred = mlp_clf.predict(new_X_test)
```

```
Iteration 1, loss = 0.51515358
Iteration 2, loss = 0.43941632
Iteration 3, loss = 0.42570626
Iteration 4, loss = 0.43331772
```

```
Iteration 5, loss = 0.40764382
Iteration 6, loss = 0.39441906
Iteration 7, loss = 0.38384440
Iteration 8, loss = 0.37244122
Iteration 9, loss = 0.36142105
Iteration 10, loss = 0.35324388
Iteration 11, loss = 0.33810572
Iteration 12, loss = 0.33745566
Iteration 13, loss = 0.31911395
Iteration 14, loss = 0.31699107
Iteration 15, loss = 0.32137112
Iteration 16, loss = 0.32485903
Iteration 17, loss = 0.31909150
Iteration 18, loss = 0.31202846
Iteration 19, loss = 0.28559606
Iteration 20, loss = 0.36002740
Iteration 21, loss = 0.31929477
Iteration 22, loss = 0.28196541
Iteration 23, loss = 0.26923592
Iteration 24, loss = 0.26142548
Iteration 25, loss = 0.28480182
Iteration 26, loss = 0.26371197
Iteration 27, loss = 0.27743039
Iteration 28, loss = 0.25938998
Iteration 29, loss = 0.24255022
Iteration 30, loss = 0.23457857
Iteration 31, loss = 0.23201297
Iteration 32, loss = 0.22786864
Iteration 33, loss = 0.21947472
Iteration 34, loss = 0.21545828
Iteration 35, loss = 0.21610703
Iteration 36, loss = 0.20792997
Iteration 37, loss = 0.21144203
Iteration 38, loss = 0.23039354
Iteration 39, loss = 0.22156244
Iteration 40, loss = 0.22882275
Iteration 41, loss = 0.33956993
Iteration 42, loss = 0.22452186
Iteration 43, loss = 0.21166971
Iteration 44, loss = 0.20159946
Iteration 45, loss = 0.21418709
Iteration 46, loss = 0.19603618
Iteration 47, loss = 0.19648629
Iteration 48, loss = 0.21354309
Iteration 49, loss = 0.18690936
Iteration 50, loss = 0.18104178
Iteration 51, loss = 0.18716763
Iteration 52, loss = 0.25230443
```

```
Iteration 53, loss = 0.18803876
Iteration 54, loss = 0.18229915
Iteration 55, loss = 0.22434972
Iteration 56, loss = 0.17428616
Iteration 57, loss = 0.17940882
Iteration 58, loss = 0.16849219
Iteration 59, loss = 0.16520146
Iteration 60, loss = 0.16377354
Iteration 61, loss = 0.17910413
Iteration 62, loss = 0.16859064
Iteration 63, loss = 0.16063379
Iteration 64, loss = 0.20140106
Iteration 65, loss = 0.22355943
Iteration 66, loss = 0.17052399
Iteration 67, loss = 0.16367491
Iteration 68, loss = 0.15560659
Iteration 69, loss = 0.33035073
Iteration 70, loss = 0.18592161
Iteration 71, loss = 0.16743508
Iteration 72, loss = 0.15996382
Iteration 73, loss = 0.21890874
Iteration 74, loss = 0.16932110
Iteration 75, loss = 0.15612446
Iteration 76, loss = 0.16724629
Iteration 77, loss = 0.15575383
Iteration 78, loss = 0.15087445
Iteration 79, loss = 0.30816145
Iteration 80, loss = 0.22119059
Iteration 81, loss = 0.24719840
Iteration 82, loss = 0.17406028
Iteration 83, loss = 0.16809002
Iteration 84, loss = 0.16255774
Iteration 85, loss = 0.20065668
Iteration 86, loss = 0.15877248
Iteration 87, loss = 0.15177169
Iteration 88, loss = 0.15704586
Iteration 89, loss = 0.14861864
Iteration 90, loss = 0.14333641
Iteration 91, loss = 0.13963085
Iteration 92, loss = 0.14870546
Iteration 93, loss = 0.13783409
Iteration 94, loss = 0.13758879
Iteration 95, loss = 0.30640821
Iteration 96, loss = 0.17582867
Iteration 97, loss = 0.21292823
Iteration 98, loss = 0.17478652
Iteration 99, loss = 0.15695555
Iteration 100, loss = 0.14857294
```

```
Iteration 101, loss = 0.14097499
Iteration 102, loss = 0.25251360
Iteration 103, loss = 0.15266756
Iteration 104, loss = 0.16144775
Iteration 105, loss = 0.14711195
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs.
Stopping.
```

[74]:
```
acc_nn = accuracy_score(new_Y_test, y_pred)
prec_nn = precision_score(new_Y_test, y_pred)
rec_nn = recall_score(new_Y_test, y_pred)
f1_nn = f1_score(new_Y_test, y_pred)

print("Accuracy:", acc_nn)
print("Precision:", prec_nn)
print("Recall:", rec_nn)
print("F1 Score:", f1_nn)
```

```
Accuracy: 0.8424657534246576
Precision: 0.14388489208633093
Recall: 0.3225806451612903
F1 Score: 0.19900497512437812
```

7.1 Hyperparameter tuning

[84]:
```
nn_clf = MLPClassifier(max_iter=200)

parameter_space = {
    'hidden_layer_sizes': [(50,100,150), (100,)],
    'activation': ['tanh', 'relu'],
    'solver': ['sgd', 'adam'],
    'alpha': [0.0001, 0.001],
    'learning_rate': ['constant','adaptive'],
}

grid_clf = GridSearchCV(nn_clf, parameter_space, n_jobs=-1, cv=3, scoring=f1)
```

[85]:
```
grid_result = grid_clf.fit(new_X_train, new_Y_train)
```

[86]:
```
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
Best: 0.924051 using {'activation': 'relu', 'alpha': 0.0001,
'hidden_layer_sizes': (50, 100, 150), 'learning_rate': 'adaptive', 'solver':
'adam'}
```

8 7. K-Fold Cross Validation

[94]:
```
from sklearn.model_selection import KFold
from sklearn import model_selection
```

8.1 Stratified K Fold

[100]:
```
kfold = model_selection.StratifiedKFold(n_splits=5)
```

[102]:
```
bag_model_kfold = BaggingClassifier(base_estimator=tree, n_estimators=6,␣
 ↪max_samples=200, bootstrap=True)

nn_model_kfold = MLPClassifier(hidden_layer_sizes=(150,100,50), max_iter=200,␣
 ↪alpha=0.001,
                     solver='adam')

bag_results_kfold = model_selection.cross_val_score(bag_model_kfold, pca_data,␣
 ↪df1_labels, cv=kfold, scoring=f1)

nn_results_kfold = model_selection.cross_val_score(nn_model_kfold, pca_data,␣
 ↪df1_labels, cv=kfold, scoring=f1)


# Because we're collecting results from all runs, we take the mean value
print(" Bagging f1 score: %.2f%%" % (bag_results_kfold.mean()*100.0))

print("Neural Network f1 score: %.2f%%" % (nn_results_kfold.mean()*100.0))
```

```
/Users/ojasbardiya/anaconda3/lib/python3.7/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:617:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
/Users/ojasbardiya/anaconda3/lib/python3.7/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:617:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
/Users/ojasbardiya/anaconda3/lib/python3.7/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:617:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

 Bagging f1 score: 50.55%
Neural Network f1 score: 53.51%
```

```
/Users/ojasbardiya/anaconda3/lib/python3.7/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:617:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
```

8.2 StratifiedShuffleSplit

[103]:
```python
kfold = model_selection.StratifiedShuffleSplit(n_splits=5, test_size=0.2,␣
 ↪random_state=20)
```

[104]:
```python
bag_model_kfold = BaggingClassifier(base_estimator=tree, n_estimators=6,␣
 ↪max_samples=200, bootstrap=True)

nn_model_kfold = MLPClassifier(hidden_layer_sizes=(150,100,50), max_iter=200,␣
 ↪alpha=0.001,
                     solver='adam')

bag_results_kfold = model_selection.cross_val_score(bag_model_kfold, pca_data,␣
 ↪df1_labels, cv=kfold, scoring=f1)

nn_results_kfold = model_selection.cross_val_score(nn_model_kfold, pca_data,␣
 ↪df1_labels, cv=kfold, scoring=f1)


# Because we're collecting results from all runs, we take the mean value
print(" Bagging f1 score: %.2f%%" % (bag_results_kfold.mean()*100.0))

print("Neural Network f1 score: %.2f%%" % (nn_results_kfold.mean()*100.0))
```

```
/Users/ojasbardiya/anaconda3/lib/python3.7/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:617:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
/Users/ojasbardiya/anaconda3/lib/python3.7/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:617:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
/Users/ojasbardiya/anaconda3/lib/python3.7/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:617:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

 Bagging f1 score: 49.84%
Neural Network f1 score: 52.10%
```

```
/Users/ojasbardiya/anaconda3/lib/python3.7/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:617:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
```

8.3 8. Custom Models

8.4 SVM

[107]:
```
clf_svm = SVC(probability=True, gamma='scale')
clf_svm.fit(new_X_train, new_Y_train)
```

[107]:
```
SVC(probability=True)
```

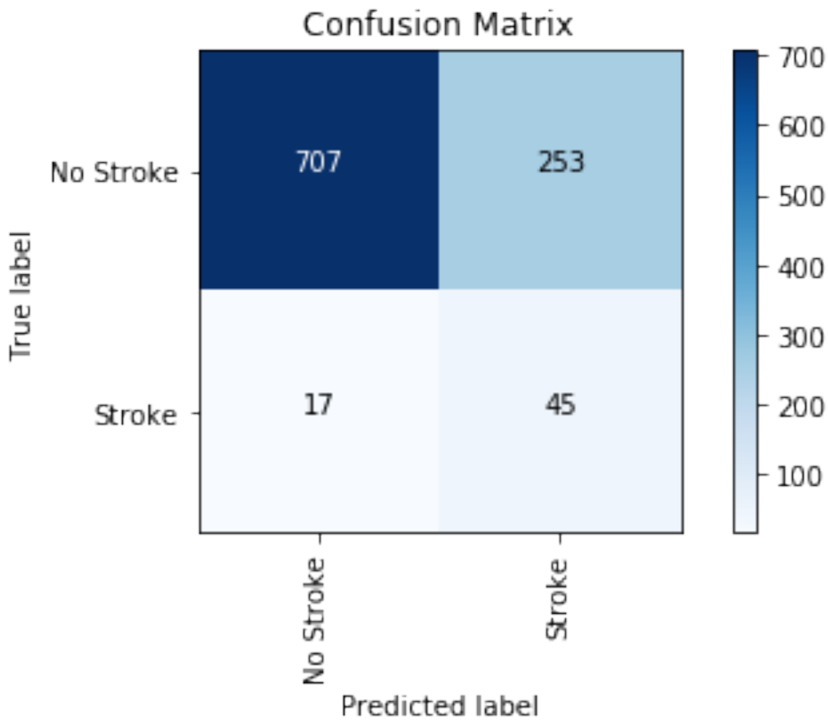
[109]:
```
y_svm_pred = clf_svm.predict(new_X_test)
```

[113]:
```
acc_svm = accuracy_score(new_Y_test, y_svm_pred)
prec_svm = precision_score(new_Y_test, y_svm_pred)
rec_svm = recall_score(new_Y_test, y_svm_pred)
f1_svm = f1_score(new_Y_test, y_svm_pred)

print("Accuracy:", acc_svm)
print("Precision:", prec_svm)
print("Recall:", rec_svm)
print("F1 Score:", f1_svm)
```

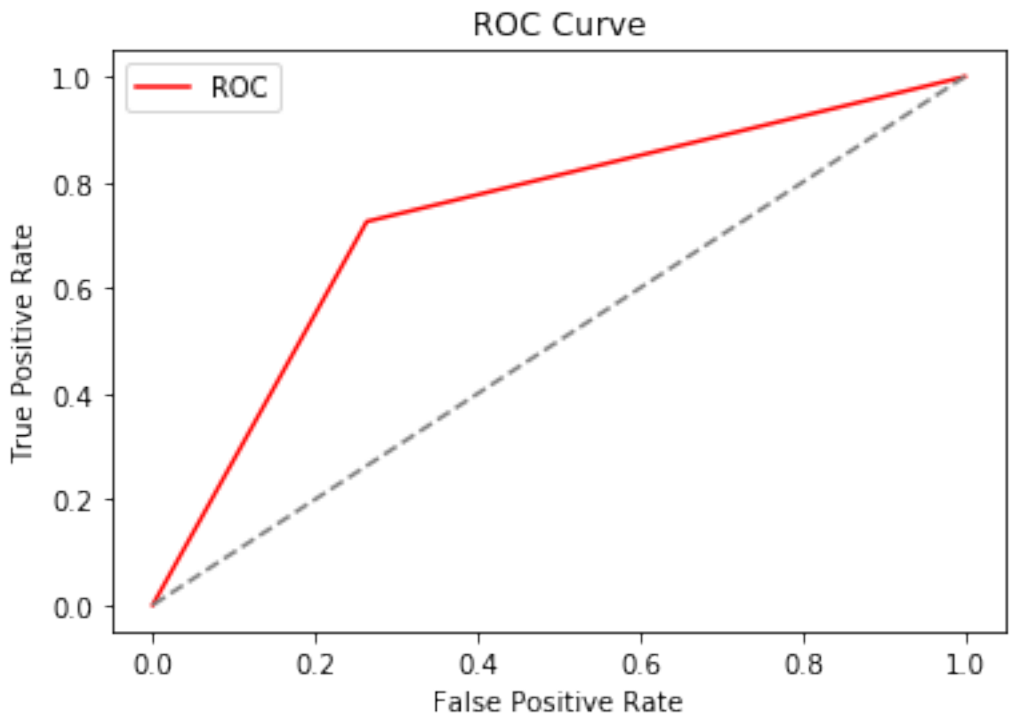
```
Accuracy: 0.735812133072407
Precision: 0.15100671140939598
Recall: 0.7258064516129032
F1 Score: 0.25
```

[114]:
```
draw_confusion_matrix(new_Y_test, y_svm_pred, ['No Stroke', 'Stroke'])
```



[115]:
```
fpr, tpr, threshold = metrics.roc_curve(new_Y_test, y_svm_pred)
plt.plot(fpr, tpr, color='red', label='ROC')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()

auc = np.trapz(tpr,fpr)
print('AUC:', auc)
```




```
AUC: 0.7311323924731183
```

8.5 Bayesian Classification

[116]:
```python
from sklearn.naive_bayes import GaussianNB
```

[117]:
```python
clf_bayes = GaussianNB().fit(new_X_train, new_Y_train)
```

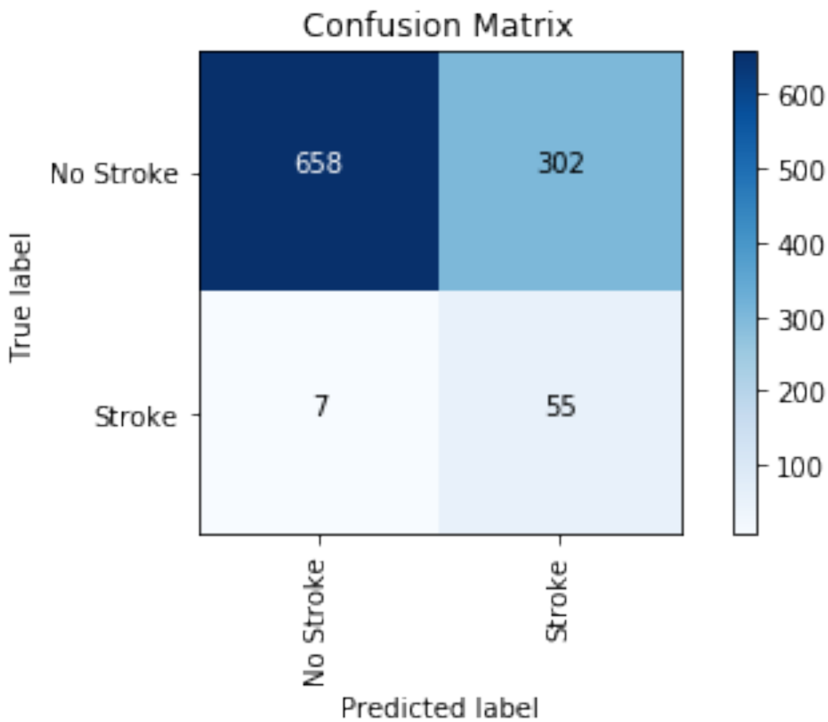
[118]:
```python
preds = clf_bayes.predict(new_X_test)
```

[119]:
```python
acc_bayes = accuracy_score(new_Y_test, preds)
prec_bayes = precision_score(new_Y_test, preds)
rec_bayes = recall_score(new_Y_test, preds)
f1_bayes = f1_score(new_Y_test, preds)

print("Accuracy:", acc_bayes)
print("Precision:", prec_bayes)
print("Recall:", rec_bayes)
print("F1 Score:", f1_bayes)
```

```
Accuracy: 0.6976516634050881
Precision: 0.15406162464985995
Recall: 0.8870967741935484
F1 Score: 0.26252983293556087
```

[120]:
```
draw_confusion_matrix(new_Y_test, preds, ['No Stroke', 'Stroke'])
```



CS M148 – Project 3 Report

1. Executive Summary

The main objective of this project is to predict the likelihood of a patient getting a stroke by employing the features present in the dataset. We build machine learning classification models and training them on the existing dataset.
The project is broadly divided into 4 components –

- Using descriptive statistics and basic visualizations to conduct a preliminary analysis of the data.
- Creating a pipeline strategy that allows imputation, augmentation, scaling and feature extraction from the dataset which can then be utilized for data modelling.
- Splitting and balancing the transformed data. Reducing dimensionality of the training data and finally building and optimizing the machine learning classification models using the transformed data. We output the following metrics for each model – accuracy, precision, recall, and F1 score.
- Cross validating the results obtained from the training data.

First, we display the mean, median and standard deviation of each numerical column feature in the dataset and obtain the correlation between different features to determine which of them should be retained for training the classification models.
Next, we determined whether the existing input parameters were numerical or categorical in nature, and in case of the latter decided whether it was ordinal or non-ordinal relationship. We then executed a pipeline that did the following –

- Augmented the dataset using a feature crossing and replaced null values with the respective column mean.
- Normalized the numerical features by scaling them in accordance with a standard normal distribution. Performed a one-hot encoding or label-encoding on the categorical features depending on whether it was ordinal or non-ordinal.

After, we split the transformed data into training and test sets and balance the data by generating synthetic samples of the minority class (occurrences of a stroke in this case). Then, we apply a Logistic Regression model and determine which features to prioritize by displaying some basic inferential statistics.
Next, we reduce the dimensionality of the data using PCA (Principal Components Analysis). We split and balance the data once again and then model the training data using the following classification strategies –

- Leveraging bagging on a decision tree classifier.
- Multi-Layer Perceptron classifier

We then optimize the following models using standard hyperparameter tuning techniques and then perform a cross-validation on our training results for the aforementioned models.
Finally, we implement 2 classification models of our own choice – an SVM (Support Vector Machine) and a Naïve Bayes Classifier.

2. Background/Introduction

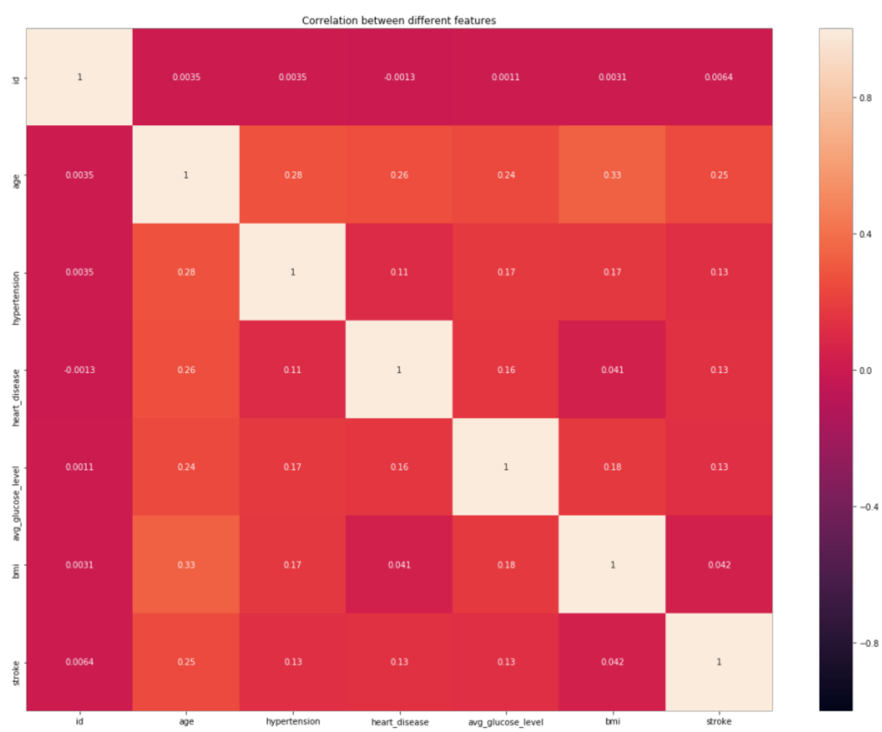
Some of the major domain challenges faced are –

- Eliminating the number of false negatives as the would reduce the usability of our model, since we would not be able predict strokes in patients accurately.
- Assessing what features are best to model the risk of a patient getting a stroke. Some features may not easily available as a consequence of patients being reluctant to disclose them. Hence, doctors and nurses may interpret statistics differently than classification models and thus there would be some discrepancy there.

While doctors would traditionally diagnose patients for risk of stroke themselves after looking at patient data, the introduction of machine learning in healthcare could provide new accuracy and efficiency in diagnosing patients for being at risk of strokes and thus work is needed to optimize these models.

3. Methodology

First, we do some basic statistical analysis - obtain the mean, median and standard deviation of each numerical column feature in the dataset and obtain the correlation between different features using a heatmap (shown below) to determine which of them should be retained for training the classification models.



Since ever_married and Residence_type are ordinal and binary categorical variables we perform a label-encoding on them prior to implementing the pipeline.
For the pipeline –

- We scale the three numerical features – bmi, average glucose level, and age – using StandardScaler() for better model performance and to ensure when we perform PCA later it does not skew towards high magnitude features.
- We augment a new feature **bmi_x_age** by multiplying the columns of bmi and age in order since those have the highest correlation amongst any two features and were positively correlated w.r.t stroke.
- Since only bmi has null values, we simply impute it with the column mean as it is the most representative of its distribution.
- The 3 non-ordinal categorical variables - smoking_status, work_type, and gender are one-hot encoded so that the impact for each unique value for each label is captured in our models.

We split the dataset into test and training sets after the pipeline transformation in order to model it.
It is also necessary to balance the data because of the skewed nature of the target label – 4861 labels correspond to having no stroke and 249 correspond to having a stroke – implying the dataset is highly imbalanced. Since using SMOTE to generate synthetic labels avoids overfitting as seen in the case with oversampling, we choose that particular method to balance it. We only balance the training data to make sure the testing set is representative of the original imbalanced data. If not, it can lead to our models being skewed and inefficient when dealing with real-world data.
Our first model is a logistic regression – we use both 'sag' and 'lbfgs' solvers to see if there is any difference in the two.
We then a perform a PCA on the transformed data obtained via the pipeline, setting the number of components parameter to 0.8 in order to obtain the minimum number of components so that 80% of the variance is retained. This is to make sure there is no overfitting. We then split and balance the data in the same manner as described previously. We perform logistic regression again and observe the difference in results.
For optimizing our results in the following 2 models, we focus on F1 score since the target class is highly imbalanced.

- For our ensemble method, we leverage bagging with a decision tree classifier and optimize it using hyperparameter tuning on the following inputs – n_estimators, max_samples, and max_features.
- For our Neural Network classifier, we use a MLP classifier and optimize it using hyperparameter tuning on the following inputs - hidden_layer_sizes, activation, solver, alpha, and learning_rate.

We then cross-validate our training results based on F1 score using 2 approaches – StratifiedShuffleSplit and StratifiedKFold – both of which ensure that the training and test sets are representative of the actual data.
Finally, we implement two of our models – SVM and a Naïve Bayes Classifier.

4. Results

Logistic Regression: (before PCA transformation)

'sag':

Accuracy: 0.7534246575342466
Precision: 0.1701388888888889
Recall: 0.7903225806451613
F1 Score: 0.27999999999999997

‘lbfgs’:
Accuracy: 0.7524461839530333
Precision: 0.1695501730103806
Recall: 0.7903225806451613
F1 Score: 0.2792022792022792

<u>Logistic Regression after PCA transformation:</u>
‘sag’:
Accuracy: 0.7397260273972602
Precision: 0.17096774193548386
Recall: 0.8548387096774194
F1 Score: 0.2849462365591398

‘lbfgs’:
Accuracy: 0.7397260273972602
Precision: 0.17096774193548386
Recall: 0.8548387096774194
F1 Score: 0.2849462365591398

<u>Bagging(Using a decision tree classifier):</u>

Non-PCA data:
Accuracy: 0.7397260273972602
Precision: 0.16883116883116883
Recall: 0.8387096774193549
F1 Score: 0.28108108108108104

PCA data:
Accuracy: 0.700587084148728
Precision: 0.1534090909090909
Recall: 0.8709677419354839
F1 Score: 0.2608695652173913

Hyperparameter tuning for F1 score:

Best: 0.936358 using {'max_features': 4, 'max_samples': 1.0, 'n_estimators': 100}

<u>MLP classifier:</u>

PCA data:
Accuracy: 0.8424657534246576
Precision: 0.14388489208633093
Recall: 0.3225806451612903
F1 Score: 0.19900497512437812

Hyperparameter tuning for F1 score:
Best: 0.924051 using {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 150), 'learning_rate': 'adaptive', 'solver': 'adam'}

<u>KFold cross-validation:</u>

StratifiedKFold:
Bagging f1 score: 50.55%
Neural Network f1 score: 53.51%

StratifiedShuffleSplit:
Bagging f1 score: 49.84%
Neural Network f1 score: 52.10%

<u>SVM:</u>

Accuracy: 0.735812133072407
Precision: 0.15100671140939598
Recall: 0.7258064516129032
F1 Score: 0.25

<u>Naïve Bayes:</u>
Accuracy: 0.6976516634050881
Precision: 0.15406162464985995
Recall: 0.8870967741935484
F1 Score: 0.26252983293556087

5. Discussion

We evaluate our models based primarily on F1 score due to the imbalanced nature of our dataset. Evaluating solely in accuracy may cause us to have skewed results towards not predicting a stroke and thus will not be useful in the medical field since we are unable to predict strokes in cases where patients actually have them.

The best model (prior to hyperparameter tuning) using this criterion is Logistic Regression using PCA-transformed data.
The best model (after hyperparameter tuning) using this criterion is Bagging while leveraging a Decision Tree Classifier using PCA-transformed data, with the parameters as in the results.
For all models, we see the F1 score is between 0.25-0.30 and the accuracy usually ranges between 0.75-0.85. This is because the dataset is highly imbalanced with few cases of stroke, so the model is able to obtain high accuracy by simply predicting a non-stroke for a lot of test cases, but the few number of labels in which stroke is incident causes the model to have a large number of false positives compared to true positives and thus a low precision and therefore a low F1 score.

<u>Recommendations for Hospitals:</u>

- The UCLA Hospital can use any of the existing models developed in the project by simply inputting the features the have to classify whether or not a patient has risk of suffering a stroke.
- Since the testing and training data were obtained after transforming the data using pipeline, the hospital may need to modify the pipeline according the features they have in order to make sure the data is transformed in such a manner so that models can interpret it. This shouldn't be too complicated – they just have to determine whether the features they input are numerical or categorical (ordinal or non-ordinal) and with some slight modifications the pipeline can implemented
- In this project, I split the transformed data before balancing it so that the testing set was representative of the original data. The data may be different in the hospital so they can reverse the two steps accordingly in order to improve the F1 score though it must be kept in mind the balancing the dataset before can lead to inefficient results on real-world data as it is not representative. If the data is balanced, then using SMOTE or any other balancing technique can be avoided.

6. Conclusion

In this project, we have highlighted the difficulty of predicting a stroke due to the imbalanced nature of the dataset as a consequence of real-world statistics but has given us relatively efficient classification models for the same. We have developed models that have high accuracy as well as high recall but low F1 score– these can be used to determine if a patient is at no risk for suffering a stroke as well as predict the incidence of stoke quite well but eschews a high number of false positives. We are able to lower the dimensionality of the training set in order to prevent the risk of overfitting due to a high number of features and eventually choose the best-suited model. We finally performed a KFold cross validation which determined that our results were generalizable to real-world data.