Healthcare Capstone project

February 24, 2024

[1]:
```
#Load necessary library
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="white", color_codes=True)
sns.set(font_scale=1.2)
```

[2]:
```
df = pd.read_csv('health care diabetes.csv')
df.head()
```

[2]:
```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6      148             72             35        0  33.6
1            1       85             66             29        0  26.6
2            8      183             64              0        0  23.3
3            1       89             66             23       94  28.1
4            0      137             40             35      168  43.1

   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.351   31        0
2                     0.672   32        1
3                     0.167   21        0
4                     2.288   33        1
```

[3]:
```
cols_with_null_as_zero = ['Glucose', 'BloodPressure', 'SkinThickness',␣
 ↪'Insulin', 'BMI']
df[cols_with_null_as_zero] = df[cols_with_null_as_zero].replace(0, np.NaN)
```

[4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   763 non-null    float64
```

```
 2   BloodPressure             733 non-null    float64
 3   SkinThickness             541 non-null    float64
 4   Insulin                   394 non-null    float64
 5   BMI                       757 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
dtypes: float64(6), int64(3)
memory usage: 54.1 KB
```

[5]:
```
df.isnull().sum()
```

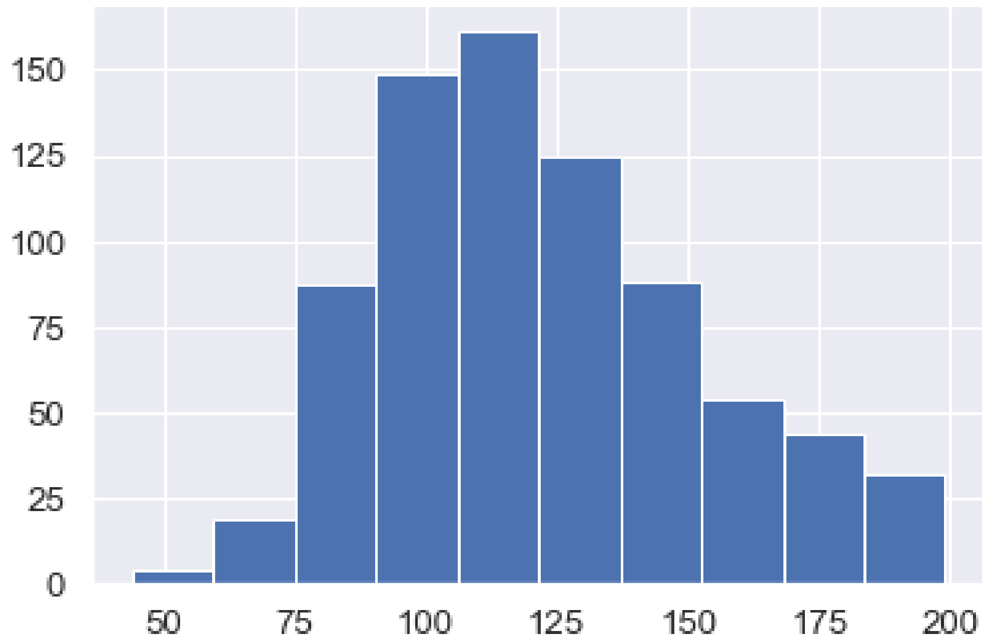
[5]:
```
Pregnancies                   0
Glucose                       5
BloodPressure                35
SkinThickness               227
Insulin                     374
BMI                          11
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64
```

[6]:
```
df.describe()
```

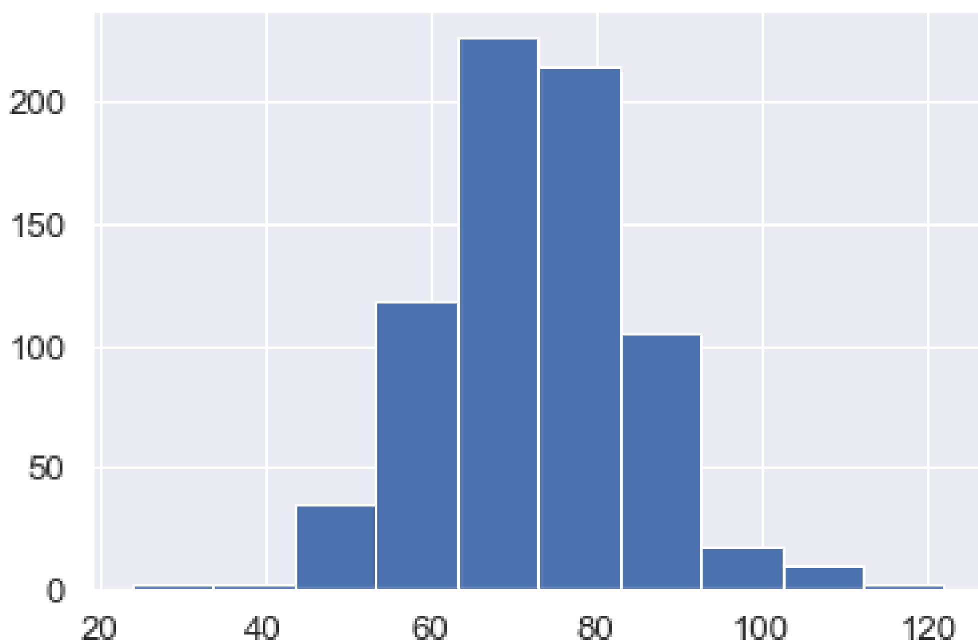
[6]:
```
       Pregnancies     Glucose  BloodPressure  SkinThickness     Insulin  \
count   768.000000  763.000000     733.000000     541.000000  394.000000
mean      3.845052  121.686763      72.405184      29.153420  155.548223
std       3.369578   30.535641      12.382158      10.476982  118.775855
min       0.000000   44.000000      24.000000       7.000000   14.000000
25%       1.000000   99.000000      64.000000      22.000000   76.250000
50%       3.000000  117.000000      72.000000      29.000000  125.000000
75%       6.000000  141.000000      80.000000      36.000000  190.000000
max      17.000000  199.000000     122.000000      99.000000  846.000000

              BMI  DiabetesPedigreeFunction         Age     Outcome
count  757.000000                768.000000  768.000000  768.000000
mean    32.457464                  0.471876   33.240885    0.348958
std      6.924988                  0.331329   11.760232    0.476951
min     18.200000                  0.078000   21.000000    0.000000
25%     27.500000                  0.243750   24.000000    0.000000
50%     32.300000                  0.372500   29.000000    0.000000
75%     36.600000                  0.626250   41.000000    1.000000
max     67.100000                  2.420000   81.000000    1.000000
```

Visually explore these variables using histograms and treat the missing values

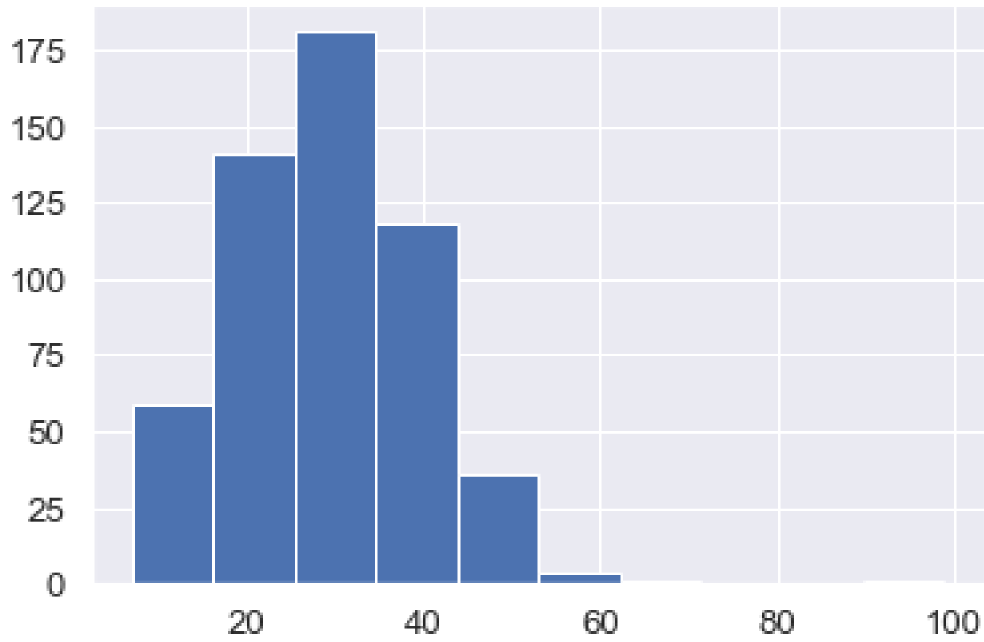
[7]:
```
df['Glucose'].hist();
```



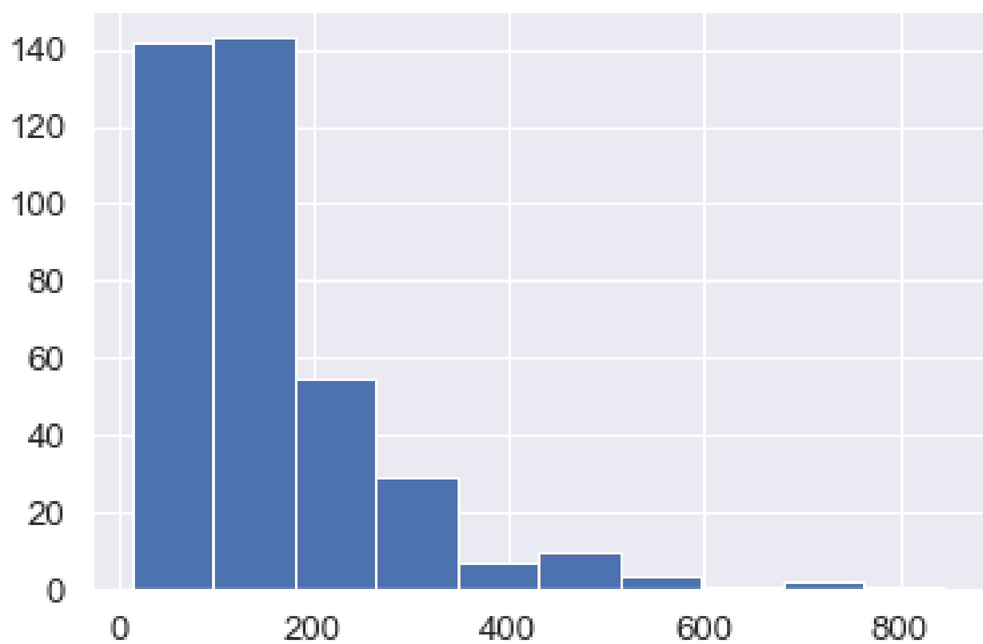

```
[8]: df['BloodPressure'].hist();
```



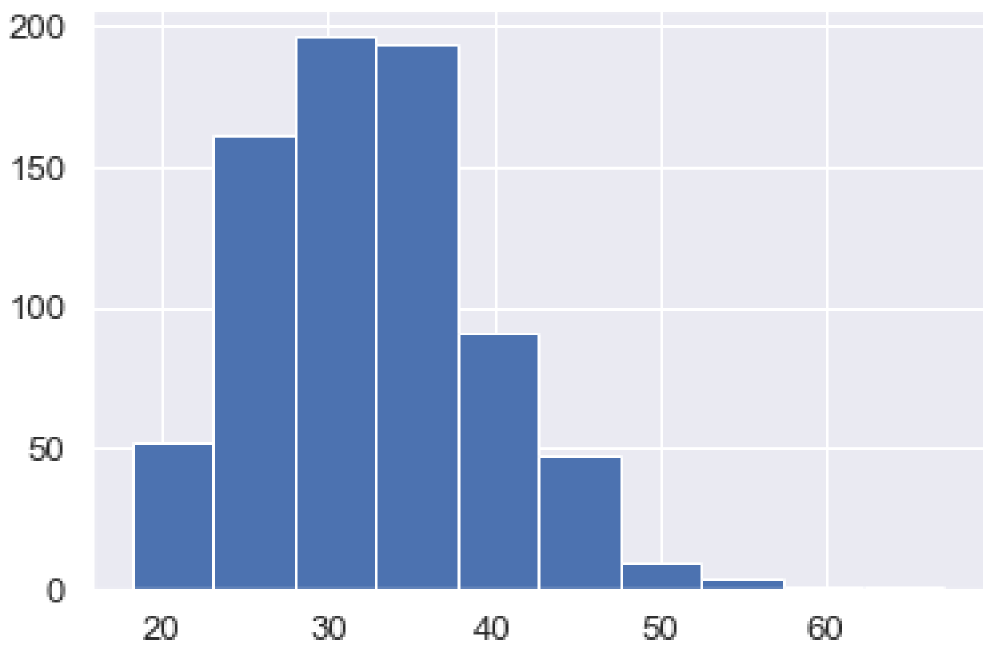

```
[9]: df['SkinThickness'].hist();
```



[10]:
```
df['Insulin'].hist();
```



[11]:
```
df['BMI'].hist();
```

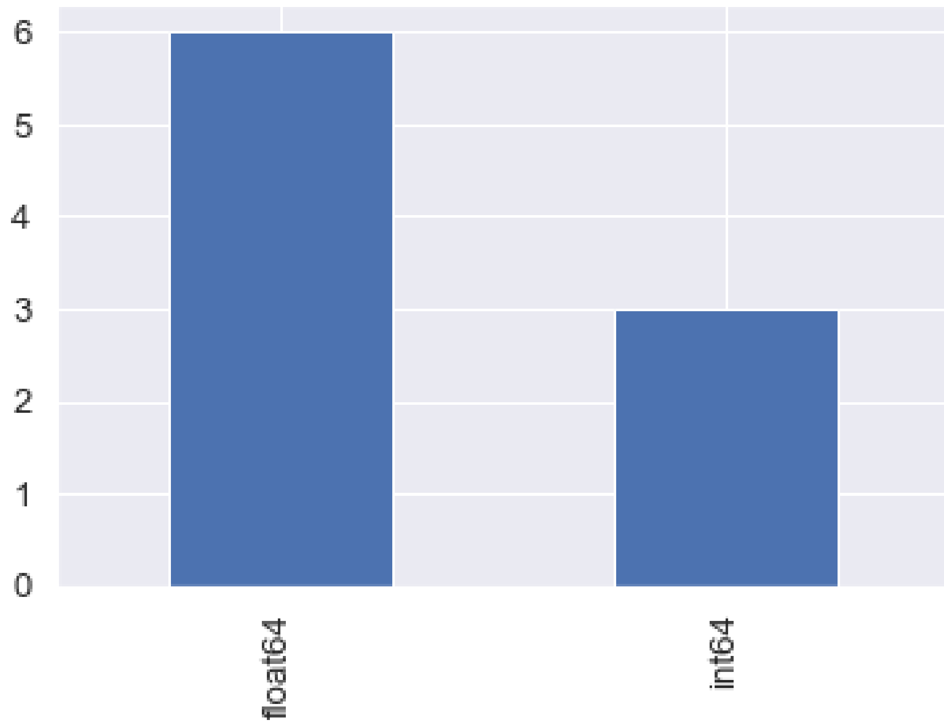


From above histograms, it is clear that **Insulin** has highly skewed data distribution * Glucose - replace missing values with mean of values. * BloodPressure - replace missing values with mean of values. * SkinThickness - replace missing values with mean of values. * Insulin - replace missing values with median of values. * BMI - replace missing values with mean of values.

[12]:
```
df['Insulin'] = df['Insulin'].fillna(df['Insulin'].median())
```

[13]:
```
cols_mean_for_null = ['Glucose', 'BloodPressure', 'SkinThickness', 'BMI']
df[cols_mean_for_null] = df[cols_mean_for_null].fillna(df[cols_mean_for_null].
 ↪mean())
```

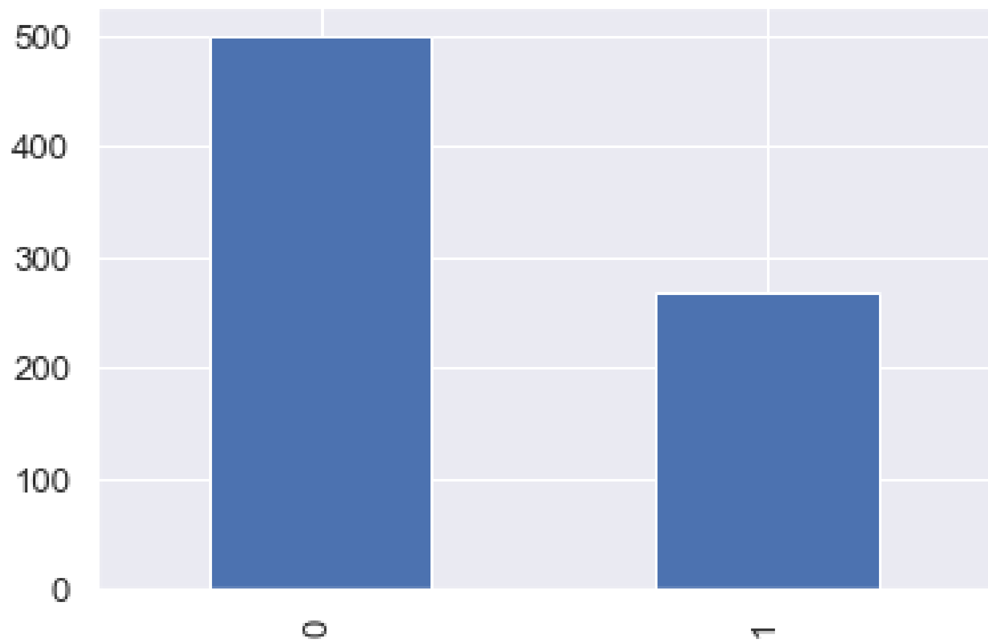
[14]:
```
df.dtypes.value_counts().plot(kind='bar');
```



Check the balance of the data by plotting the count of outcomes by their value. Describe your findings and plan future course of action:

[15]:
```
df['Outcome'].value_counts().plot(kind='bar')
df['Outcome'].value_counts()
```

[15]:
```
0    500
1    268
Name: Outcome, dtype: int64
```



Since classes in **Outcome** is little skewed so we will generate new samples using **SMOTE (Synthetic Minority Oversampling Technique)** for the class '**1**' which is under-represented in our data. We will use SMOTE out of many other techniques available since:

[16]:
```
df_X = df.drop('Outcome', axis=1)
df_y = df['Outcome']
print(df_X.shape, df_y.shape)
```

```
(768, 8) (768,)
```

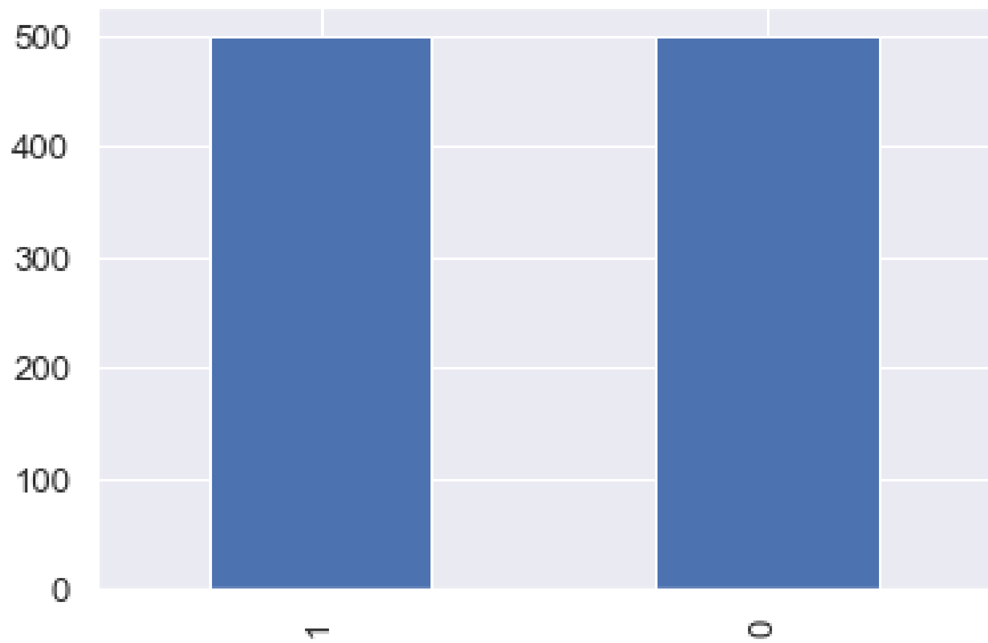
[17]:
```
from imblearn.over_sampling import SMOTE
```

[18]:
```
df_X_resampled, df_y_resampled = SMOTE(random_state=108).fit_resample(df_X,␣
    ↪df_y)
print(df_X_resampled.shape, df_y_resampled.shape)
```

```
(1000, 8) (1000,)
```

[19]:
```
df_y_resampled.value_counts().plot(kind='bar')
df_y_resampled.value_counts()
```

[19]:
```
1    500
0    500
Name: Outcome, dtype: int64
```



(2) Create scatter charts between the pair of variables to understand the relationships. Describe your findings:

[20]:
```
df_resampled = pd.concat([df_X_resampled, df_y_resampled], axis=1)
df_resampled
```

[20]:

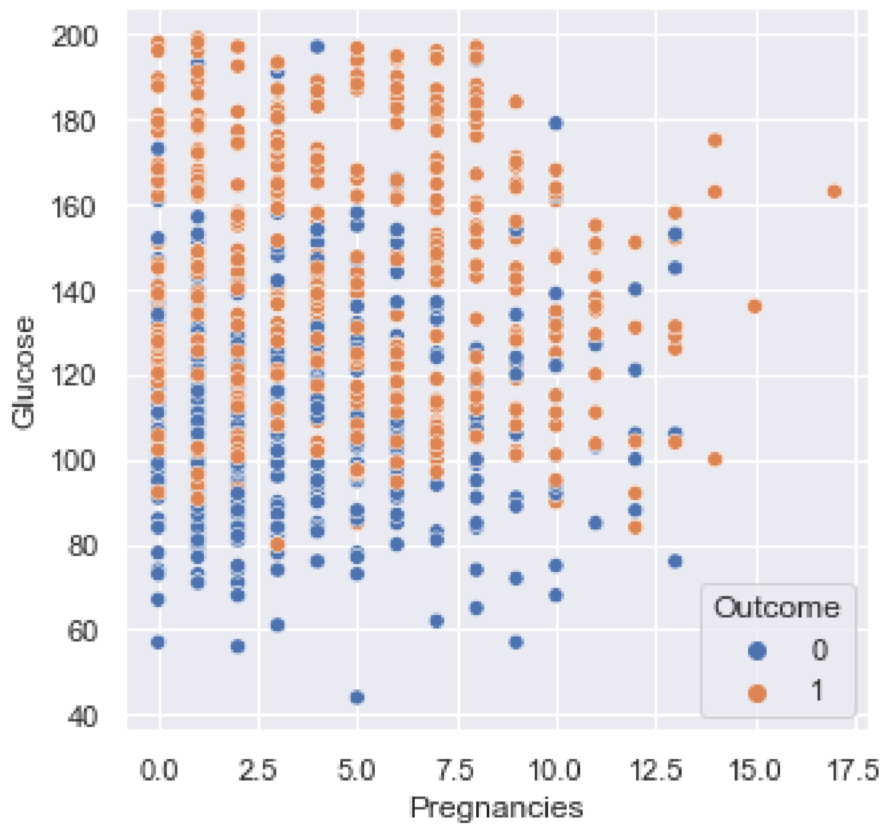
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin
0	6	148.000000	72.000000	35.000000	125.000000
1	1	85.000000	66.000000	29.000000	125.000000
2	8	183.000000	64.000000	29.153420	125.000000
3	1	89.000000	66.000000	23.000000	94.000000
4	0	137.000000	40.000000	35.000000	168.000000
..	…	…	…	…	…
995	3	164.686765	74.249021	29.153420	125.000000
996	0	138.913540	69.022720	27.713033	127.283849
997	10	131.497740	66.331574	33.149837	125.000000
998	0	105.571347	83.238205	29.153420	125.000000
999	0	127.727025	108.908879	44.468195	129.545366

\

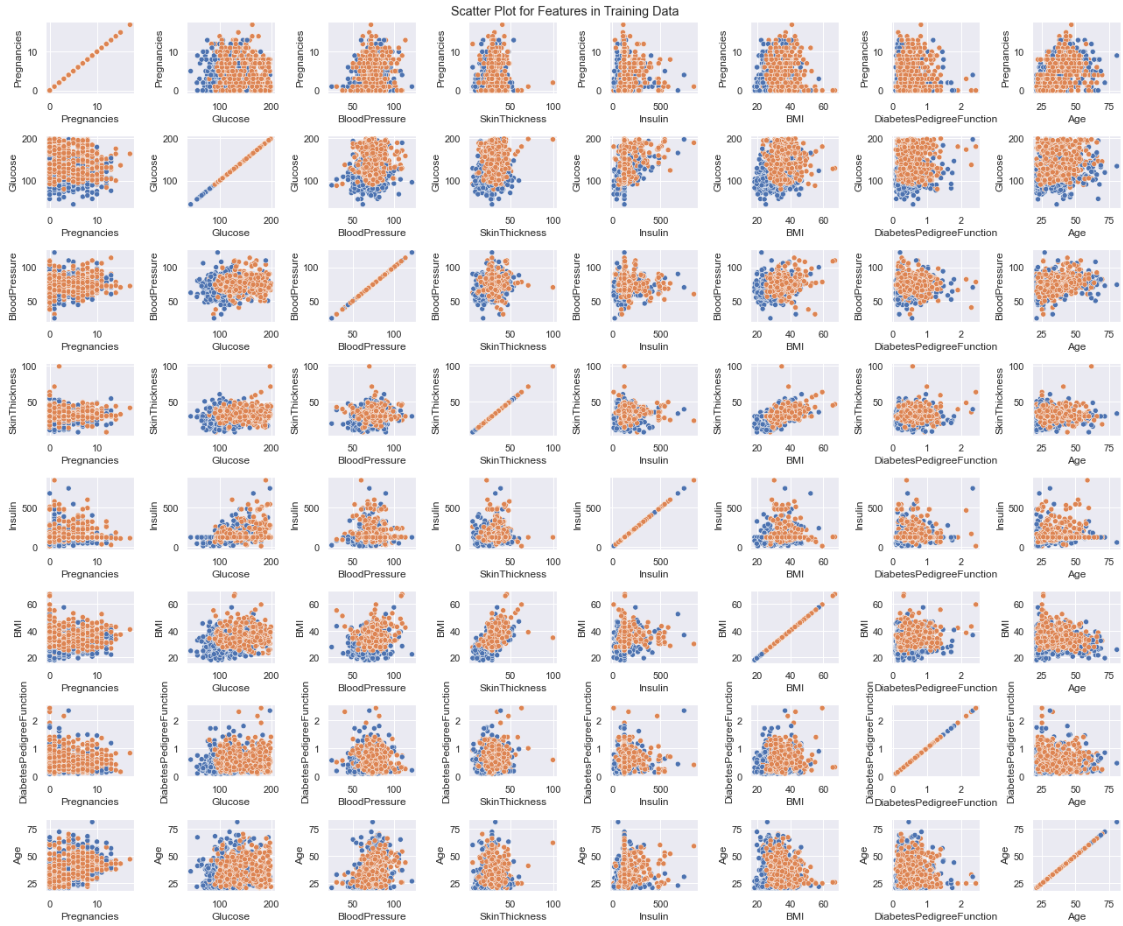
	BMI	DiabetesPedigreeFunction	Age	Outcome
0	33.600000	0.627000	50	1
1	26.600000	0.351000	31	0
2	23.300000	0.672000	32	1
3	28.100000	0.167000	21	0
4	43.100000	2.288000	33	1
..	…	…	…	…

```
995  42.767110                    0.726091   29          1
996  39.177649                    0.703702   24          1
997  45.820819                    0.498032   38          1
998  27.728596                    0.649204   60          1
999  65.808840                    0.308998   26          1

[1000 rows x 9 columns]
```

[21]:
```python
sns.set(rc={'figure.figsize':(5,5)})
sns.scatterplot(x="Pregnancies", y="Glucose", data=df_resampled, hue="Outcome");
```



[22]:
```python
fig, axes = plt.subplots(8, 8, figsize=(18, 15))
fig.suptitle('Scatter Plot for Features in Training Data')

for i, col_y in enumerate(df_X_resampled.columns):
    for j, col_x in enumerate(df_X_resampled.columns):
        sns.scatterplot(ax=axes[i, j], x=col_x, y=col_y, data=df_resampled,␣
 ↪hue="Outcome", legend = False)

plt.tight_layout()
```



Perform correlation analysis. Visually explore it using a heat map:

[23]:
```
df_X_resampled.corr()
```

[23]:

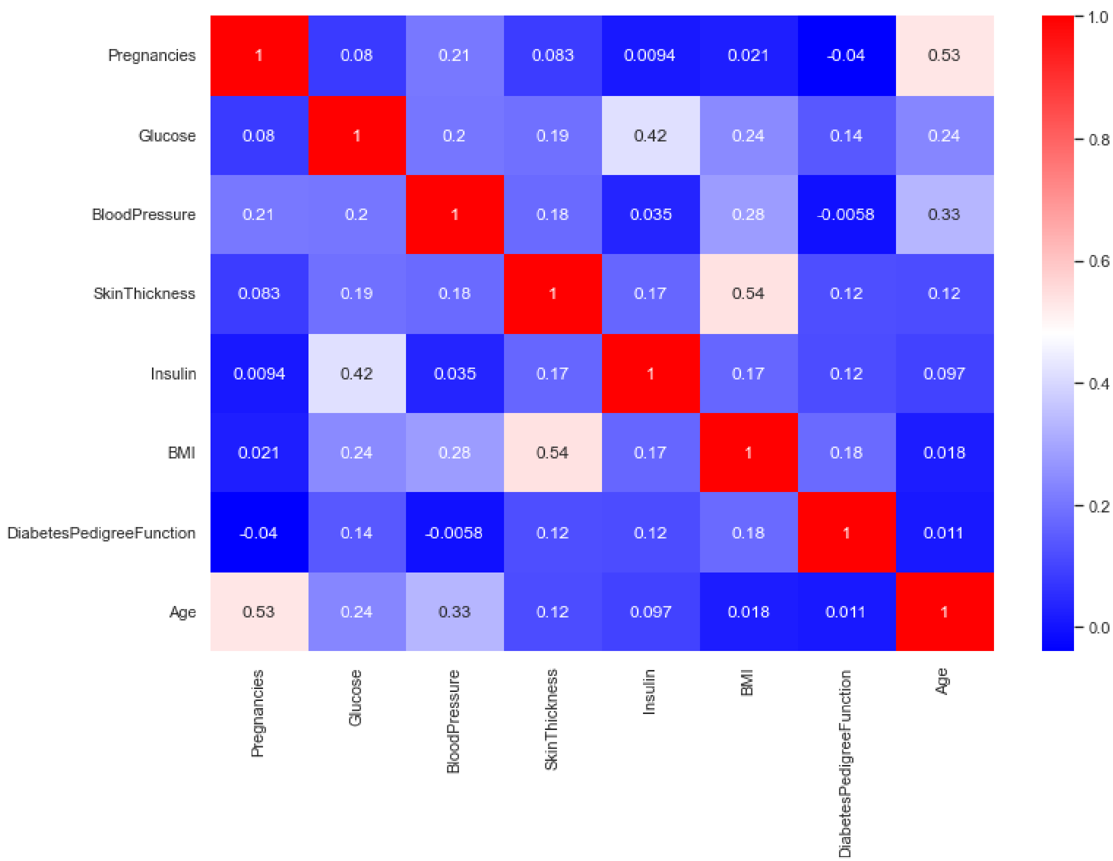
	Pregnancies	Glucose	BloodPressure	SkinThickness
Pregnancies	1.000000	0.079953	0.205232	0.082752
Glucose	0.079953	1.000000	0.200717	0.189776
BloodPressure	0.205232	0.200717	1.000000	0.176496
SkinThickness	0.082752	0.189776	0.176496	1.000000
Insulin	0.009365	0.418830	0.034861	0.170719
BMI	0.021006	0.242501	0.277565	0.538207
DiabetesPedigreeFunction	-0.040210	0.138945	-0.005850	0.120799
Age	0.532660	0.235522	0.332015	0.117644

	Insulin	BMI	DiabetesPedigreeFunction
Pregnancies	0.009365	0.021006	-0.040210
Glucose	0.418830	0.242501	0.138945
BloodPressure	0.034861	0.277565	-0.005850
SkinThickness	0.170719	0.538207	0.120799

```
Insulin                   1.000000  0.168702                  0.115187
BMI                       0.168702  1.000000                  0.177915
DiabetesPedigreeFunction  0.115187  0.177915                  1.000000
Age                       0.096940  0.017529                  0.010532

                               Age
Pregnancies               0.532660
Glucose                   0.235522
BloodPressure             0.332015
SkinThickness             0.117644
Insulin                   0.096940
BMI                       0.017529
DiabetesPedigreeFunction  0.010532
Age                       1.000000
```

[24]:

```python
plt.figure(figsize=(12,8))
sns.heatmap(df_X_resampled.corr(), cmap='bwr', annot=True);
```



[25]:

```python
from sklearn.model_selection import train_test_split, KFold, RandomizedSearchCV
```

```
from sklearn.metrics import accuracy_score, average_precision_score, f1_score,␣
↪confusion_matrix, classification_report, auc, roc_curve, roc_auc_score,␣
↪precision_recall_curve
```

[26]:
```
X_train, X_test, y_train, y_test = train_test_split(df_X_resampled,␣
↪df_y_resampled, test_size=0.15, random_state =10)
```

[27]:
```
X_train.shape, X_test.shape
```

[27]: ((850, 8), (150, 8))

[28]:
```
models = []
model_accuracy = []
model_f1 = []
model_auc = []
```

Logistic Regression:

[29]:
```
from sklearn.linear_model import LogisticRegression
lr1 = LogisticRegression(max_iter=300)
```

[30]:
```
lr1.fit(X_train,y_train)
```

[30]: LogisticRegression(max_iter=300)

[31]:
```
lr1.score(X_train,y_train)
```

[31]: 0.7294117647058823

[32]:
```
lr1.score(X_test, y_test)
```

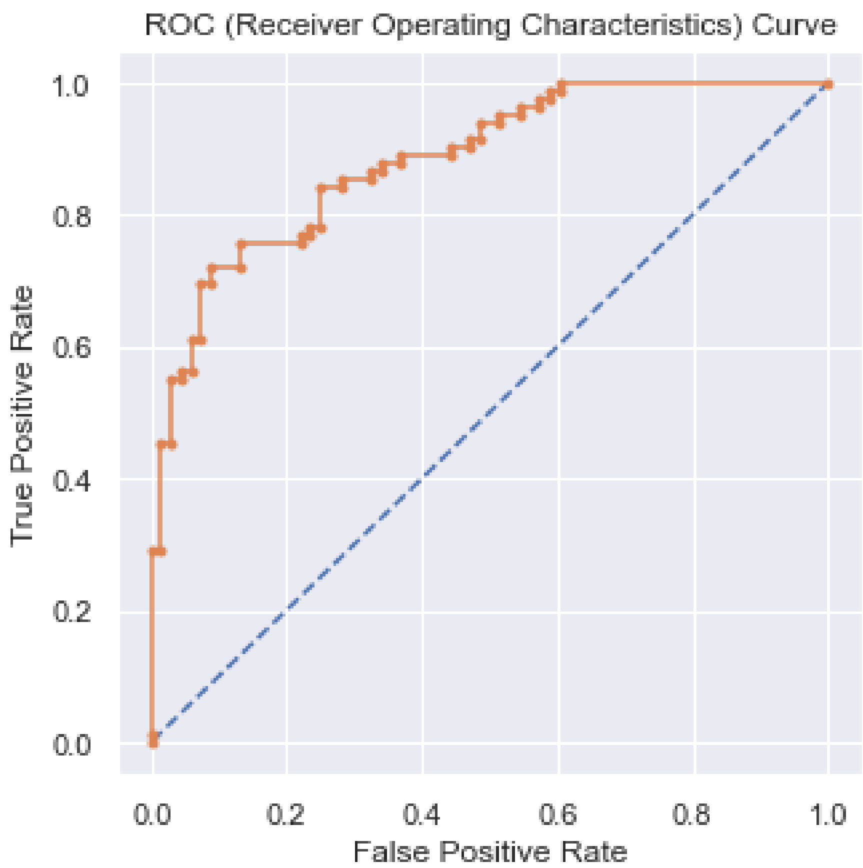
[32]: 0.76

Performance evaluation and optimizing parameters using GridSearchCV: Logistic regression does not really have any critical hyperparameters to tune. However we will try to optimize one of its parameters 'C' with the help of GridSearchCV. So we have set this parameter as a list of values form which GridSearchCV will select the best value of parameter.

[42]:
```
# Preparing ROC Curve (Receiver Operating Characteristics Curve)

probs = lr2.predict_proba(X_test)                # predict probabilities
probs = probs[:, 1]                              # keep probabilities for the␣
↪positive outcome only

auc_lr = roc_auc_score(y_test, probs)            # calculate AUC
print('AUC: %.3f' %auc_lr)
fpr, tpr, thresholds = roc_curve(y_test, probs)  # calculate roc curve
plt.plot([0, 1], [0, 1], linestyle='--')         # plot no skill
```

```
plt.plot(fpr, tpr, marker='.')                      # plot the roc curve for the␣
 ↪model
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC (Receiver Operating Characteristics) Curve");
```

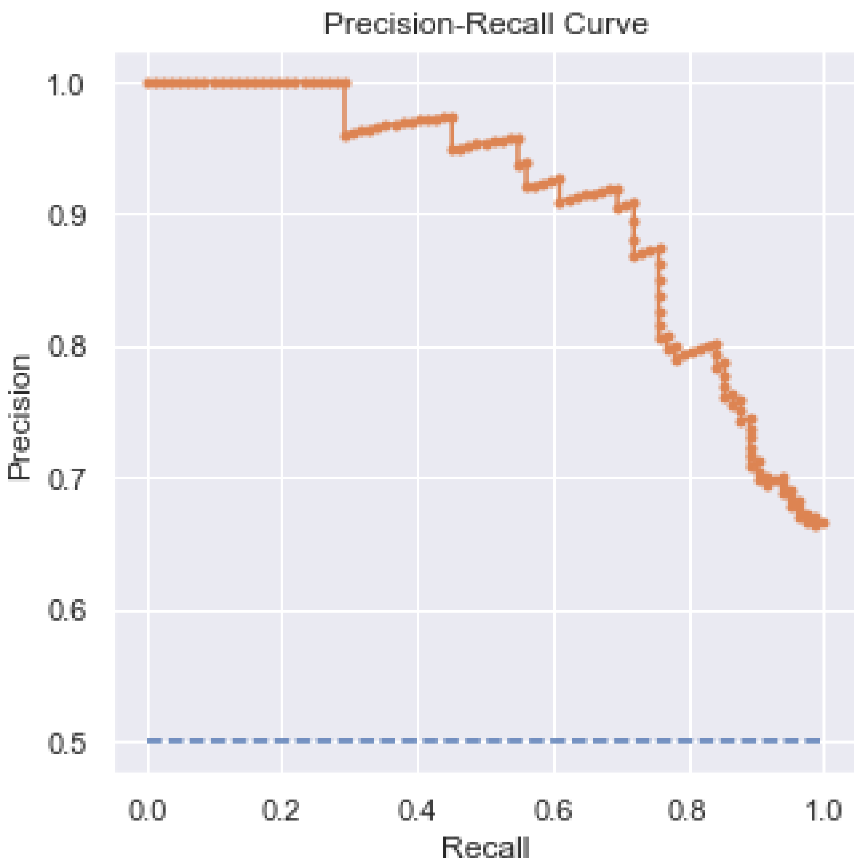
```
AUC: 0.884
```



[43]:
```
# Precision Recall Curve

pred_y_test = lr2.predict(X_test)                                         # predict␣
 ↪class values
precision, recall, thresholds = precision_recall_curve(y_test, probs) #␣
 ↪calculate precision-recall curve
f1 = f1_score(y_test, pred_y_test)                                        #␣
 ↪calculate F1 score
auc_lr_pr = auc(recall, precision)                                        #␣
 ↪calculate precision-recall AUC
```

```
ap = average_precision_score(y_test, probs)                                  #␣
  ↪calculate average precision score
print('f1=%.3f auc_pr=%.3f ap=%.3f' % (f1, auc_lr_pr, ap))
plt.plot([0, 1], [0.5, 0.5], linestyle='--')                                 # plot no␣
  ↪skill
plt.plot(recall, precision, marker='.')                                      # plot␣
  ↪the precision-recall curve for the model
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve");
```

```
f1=0.790 auc_pr=0.908 ap=0.909
```



[44]:
```
models.append('LR')
model_accuracy.append(accuracy_score(y_test, pred_y_test))
model_f1.append(f1)
model_auc.append(auc_lr)
```

Decision Tree:

[45]:
```
from sklearn.tree import DecisionTreeClassifier
dt1 = DecisionTreeClassifier(random_state=0)
```

[46]:
```
dt1.fit(X_train,y_train)
```

[46]:
```
DecisionTreeClassifier(random_state=0)
```

[47]:
```
dt1.score(X_train,y_train)           # Decision Tree always 100% accuracy over␣
 ↪train data
```

[47]:
```
1.0
```

[48]:
```
dt1.score(X_test, y_test)
```

[48]:
```
0.7733333333333333
```

Performance evaluation and optimizing parameters using GridSearchCV:

[49]:
```
parameters = {
    'max_depth':[1,2,3,4,5,None]
}
```

[50]:
```
gs_dt = GridSearchCV(dt1, param_grid = parameters, cv=5, verbose=0)
gs_dt.fit(df_X_resampled, df_y_resampled)
```

[50]:
```
GridSearchCV(cv=5, estimator=DecisionTreeClassifier(random_state=0),
             param_grid={'max_depth': [1, 2, 3, 4, 5, None]})
```

[51]:
```
gs_dt.best_params_
```

[51]:
```
{'max_depth': 4}
```

[52]:
```
gs_dt.best_score_
```

[52]:
```
0.76
```

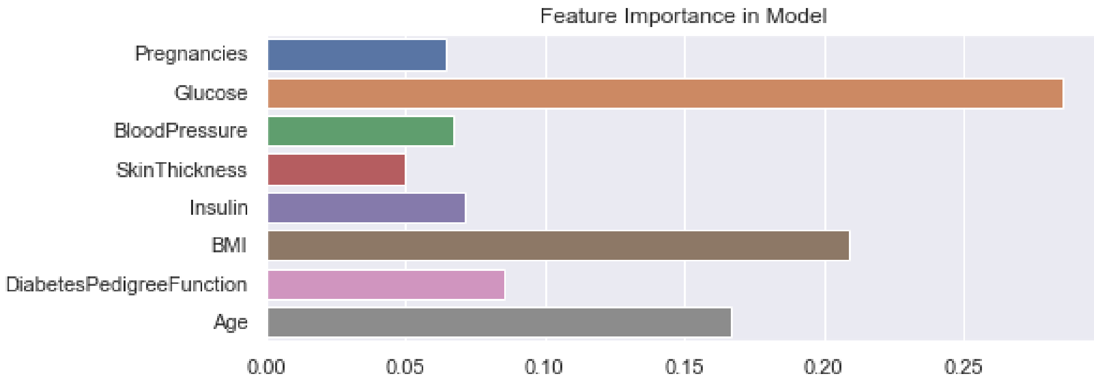
[53]:
```
dt1.feature_importances_
```

[53]:
```
array([0.06452226, 0.28556999, 0.06715314, 0.04979714, 0.07150365,
       0.20905992, 0.08573109, 0.16666279])
```

[54]:
```
X_train.columns
```

[54]:
```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age'],
      dtype='object')
```

[55]:
```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(8,3))
sns.barplot(y=X_train.columns, x=dt1.feature_importances_)
plt.title("Feature Importance in Model");
```



[56]:
```
dt2 = DecisionTreeClassifier(max_depth=4)
```

[57]:
```
dt2.fit(X_train,y_train)
```

[57]: DecisionTreeClassifier(max_depth=4)

[58]:
```
dt2.score(X_train,y_train)
```

[58]: 0.8070588235294117

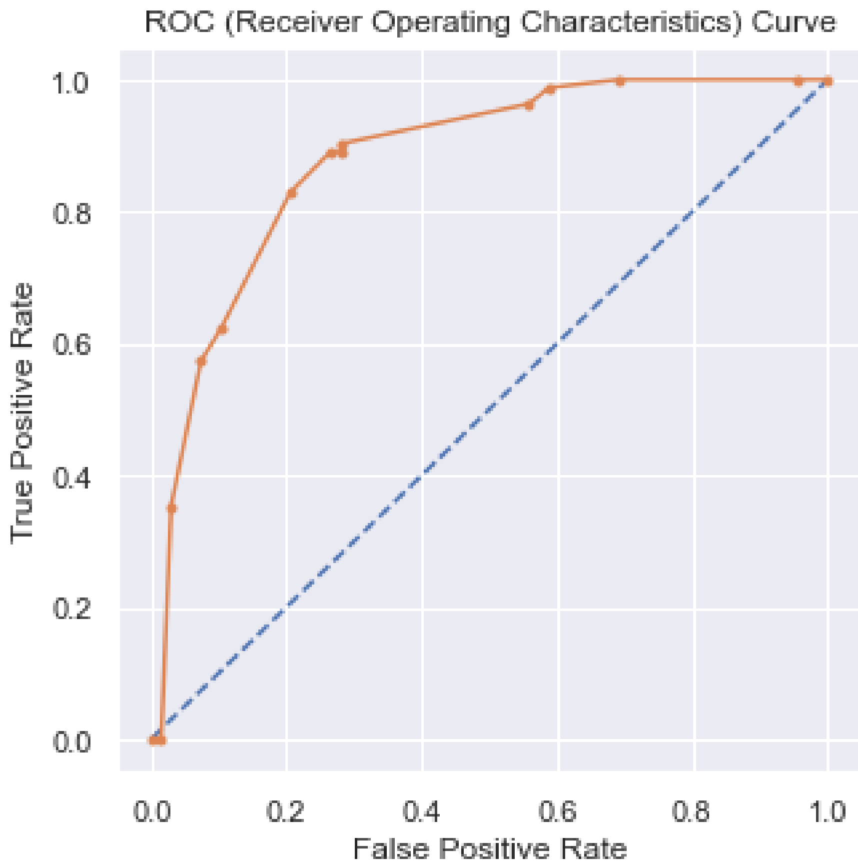
[59]:
```
dt2.score(X_test, y_test)
```

[59]: 0.82

[60]:
```
# Preparing ROC Curve (Receiver Operating Characteristics Curve)

probs = dt2.predict_proba(X_test)              # predict probabilities
probs = probs[:, 1]                            # keep probabilities for the␣
 ↪positive outcome only

auc_dt = roc_auc_score(y_test, probs)          # calculate AUC
print('AUC: %.3f' %auc_dt)
fpr, tpr, thresholds = roc_curve(y_test, probs)  # calculate roc curve
plt.plot([0, 1], [0, 1], linestyle='--')       # plot no skill
plt.plot(fpr, tpr, marker='.')                 # plot the roc curve for the␣
 ↪model
```

```
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC (Receiver Operating Characteristics) Curve");
```

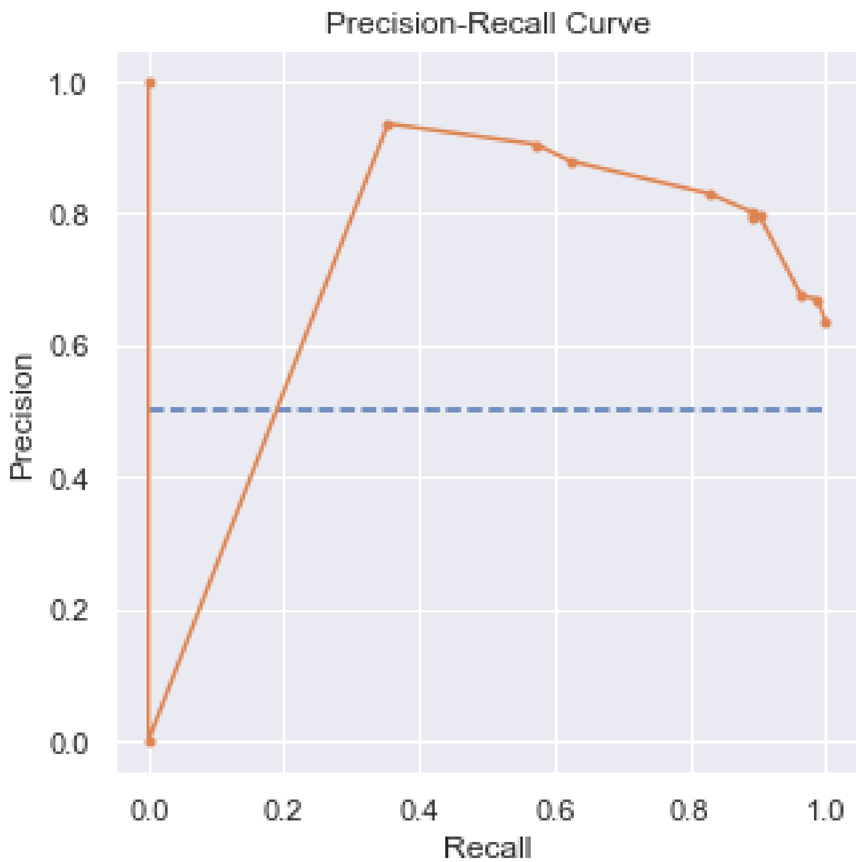
```
AUC: 0.879
```



[61]:
```
# Precision Recall Curve

pred_y_test = dt2.predict(X_test)                                   # predict␣
 ↪class values
precision, recall, thresholds = precision_recall_curve(y_test, probs) #␣
 ↪calculate precision-recall curve
f1 = f1_score(y_test, pred_y_test)                                  #␣
 ↪calculate F1 score
auc_dt_pr = auc(recall, precision)                                  #␣
 ↪calculate precision-recall AUC
ap = average_precision_score(y_test, probs)                         #␣
 ↪calculate average precision score
print('f1=%.3f auc_pr=%.3f ap=%.3f' % (f1, auc_dt_pr, ap))
```

```
plt.plot([0, 1], [0.5, 0.5], linestyle='--')                          # plot no␣
 ↪skill
plt.plot(recall, precision, marker='.')                                 # plot␣
 ↪the precision-recall curve for the model
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve");
```

```
f1=0.844 auc_pr=0.717 ap=0.868
```



[62]:
```
models.append('DT')
model_accuracy.append(accuracy_score(y_test, pred_y_test))
model_f1.append(f1)
model_auc.append(auc_dt)
```

RandomForest Classifier

[63]:
```
from sklearn.ensemble import RandomForestClassifier
rf1 = RandomForestClassifier()
```

[64]:
```
rf1 = RandomForestClassifier(random_state=0)
```

[65]:
```
rf1.fit(X_train, y_train)
```

[65]:
```
RandomForestClassifier(random_state=0)
```

[66]:
```
rf1.score(X_train, y_train)          # Random Forest also 100% accuracy over␣
 ↪train data always
```

[66]:
```
1.0
```

[67]:
```
rf1.score(X_test, y_test)
```

[67]:
```
0.8466666666666667
```

Performance evaluation and optimizing parameters using GridSearchCV:

[68]:
```
parameters = {
    'n_estimators': [50,100,150],
    'max_depth': [None,1,3,5,7],
    'min_samples_leaf': [1,3,5]
}
```

[69]:
```
gs_dt = GridSearchCV(estimator=rf1, param_grid=parameters, cv=5, verbose=0)
gs_dt.fit(df_X_resampled, df_y_resampled)
```

[69]:
```
GridSearchCV(cv=5, estimator=RandomForestClassifier(random_state=0),
             param_grid={'max_depth': [None, 1, 3, 5, 7],
                         'min_samples_leaf': [1, 3, 5],
                         'n_estimators': [50, 100, 150]})
```

[70]:
```
gs_dt.best_params_
```

[70]:
```
{'max_depth': None, 'min_samples_leaf': 1, 'n_estimators': 100}
```

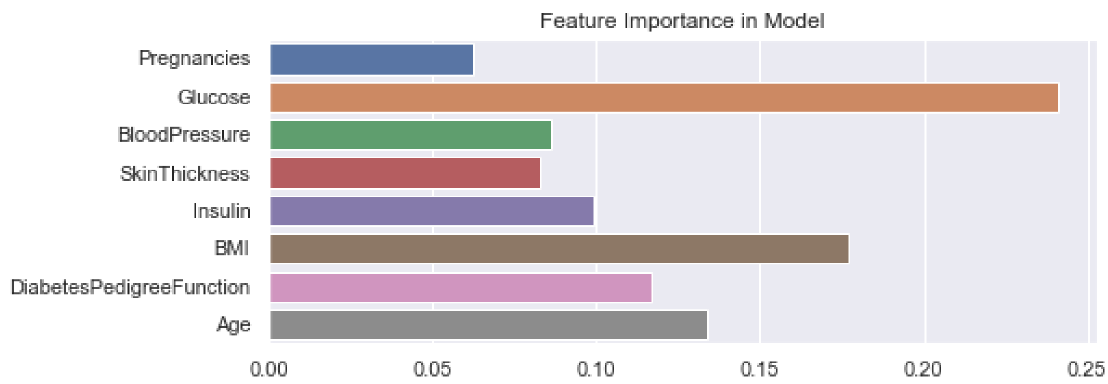
[71]:
```
gs_dt.best_score_
```

[71]:
```
0.813
```

[72]:
```
rf1.feature_importances_
```

[72]:
```
array([0.06264995, 0.24106573, 0.08653626, 0.08301549, 0.09945063,
       0.17678287, 0.11685244, 0.13364664])
```

[73]:
```
plt.figure(figsize=(8,3))
sns.barplot(y=X_train.columns, x=rf1.feature_importances_);
plt.title("Feature Importance in Model");
```



[74]:
```
rf2 = RandomForestClassifier(max_depth=None, min_samples_leaf=1,␣
 ↪n_estimators=100)
```

[75]:
```
rf2.fit(X_train,y_train)
```

[75]:
```
RandomForestClassifier()
```

[76]:
```
rf2.score(X_train,y_train)
```

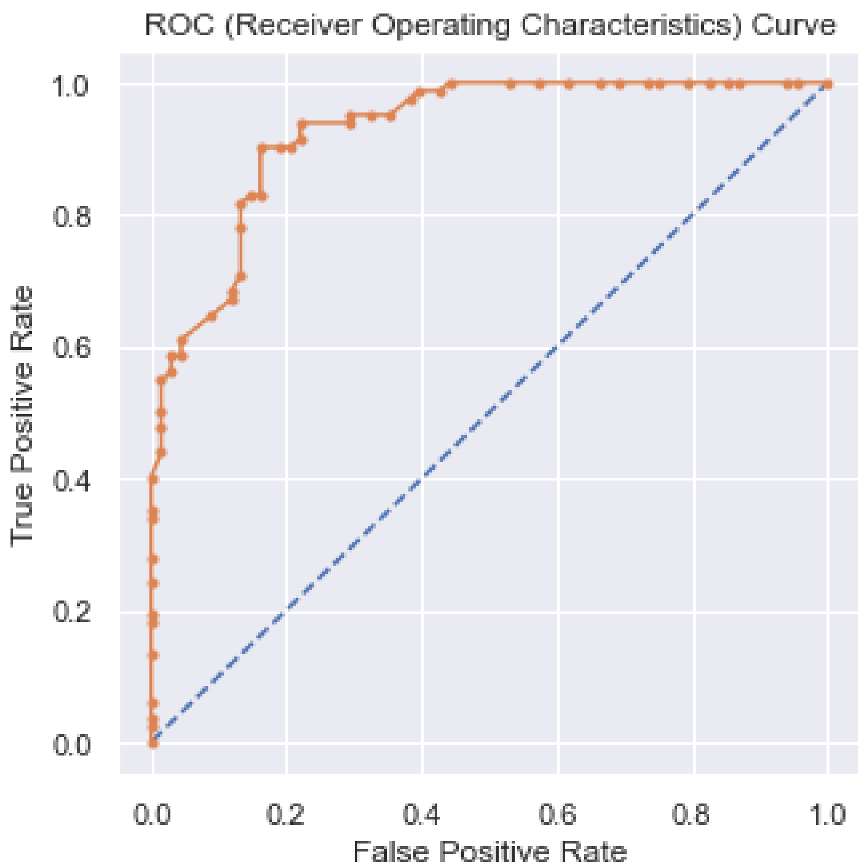
[76]:
```
1.0
```

[77]:
```
rf2.score(X_test, y_test)
```

[77]:
```
0.86
```

[78]:
```
# Preparing ROC Curve (Receiver Operating Characteristics Curve)

probs = rf2.predict_proba(X_test)               # predict probabilities
probs = probs[:, 1]                             # keep probabilities for the␣
 ↪positive outcome only

auc_rf = roc_auc_score(y_test, probs)           # calculate AUC
print('AUC: %.3f' %auc_rf)
fpr, tpr, thresholds = roc_curve(y_test, probs)  # calculate roc curve
plt.plot([0, 1], [0, 1], linestyle='--')        # plot no skill
plt.plot(fpr, tpr, marker='.')                  # plot the roc curve for the␣
 ↪model
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC (Receiver Operating Characteristics) Curve");
```

```
AUC: 0.928
```

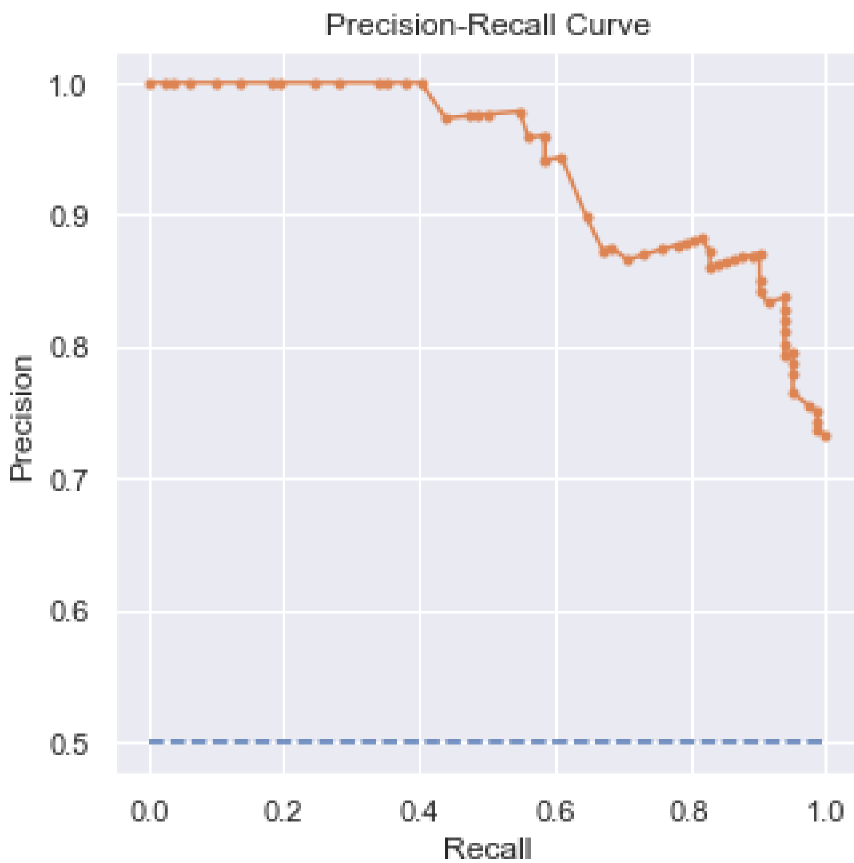


[79]:

```
# Precision Recall Curve

pred_y_test = rf2.predict(X_test)                                       # predict␣
 ↪class values
precision, recall, thresholds = precision_recall_curve(y_test, probs) #␣
 ↪calculate precision-recall curve
f1 = f1_score(y_test, pred_y_test)                                      #␣
 ↪calculate F1 score
auc_rf_pr = auc(recall, precision)                                      #␣
 ↪calculate precision-recall AUC
ap = average_precision_score(y_test, probs)                             #␣
 ↪calculate average precision score
print('f1=%.3f auc_pr=%.3f ap=%.3f' % (f1, auc_rf_pr, ap))
plt.plot([0, 1], [0.5, 0.5], linestyle='--')                           # plot no␣
 ↪skill
plt.plot(recall, precision, marker='.')                                 # plot␣
 ↪the precision-recall curve for the model
plt.xlabel("Recall")
plt.ylabel("Precision")
```

```
plt.title("Precision-Recall Curve");
```

```
f1=0.873 auc_pr=0.938 ap=0.936
```



[80]:
```
models.append('RF')
model_accuracy.append(accuracy_score(y_test, pred_y_test))
model_f1.append(f1)
model_auc.append(auc_dt)
```

K-Nearest Neighbour (KNN) Classification:

[81]:
```
from sklearn.neighbors import KNeighborsClassifier
knn1 = KNeighborsClassifier(n_neighbors=3)
```

[82]:
```
knn1.fit(X_train, y_train)
```

[82]:
```
KNeighborsClassifier(n_neighbors=3)
```

[83]:
```
knn1.score(X_train,y_train)
```

[83]: 0.8835294117647059

[84]:
```python
knn1.score(X_test,y_test)
```

[84]: 0.7866666666666666

Performance evaluation and optimizing parameters using GridSearchCV:

[85]:
```python
knn_neighbors = [i for i in range(2,16)]
parameters = {
    'n_neighbors': knn_neighbors
}
```

[86]:
```python
gs_knn = GridSearchCV(estimator=knn1, param_grid=parameters, cv=5, verbose=0)
gs_knn.fit(df_X_resampled, df_y_resampled)
```

[86]:
```
GridSearchCV(cv=5, estimator=KNeighborsClassifier(n_neighbors=3),
             param_grid={'n_neighbors': [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                                         14, 15]})
```

[87]:
```python
gs_knn.best_params_
```

[87]: {'n_neighbors': 3}

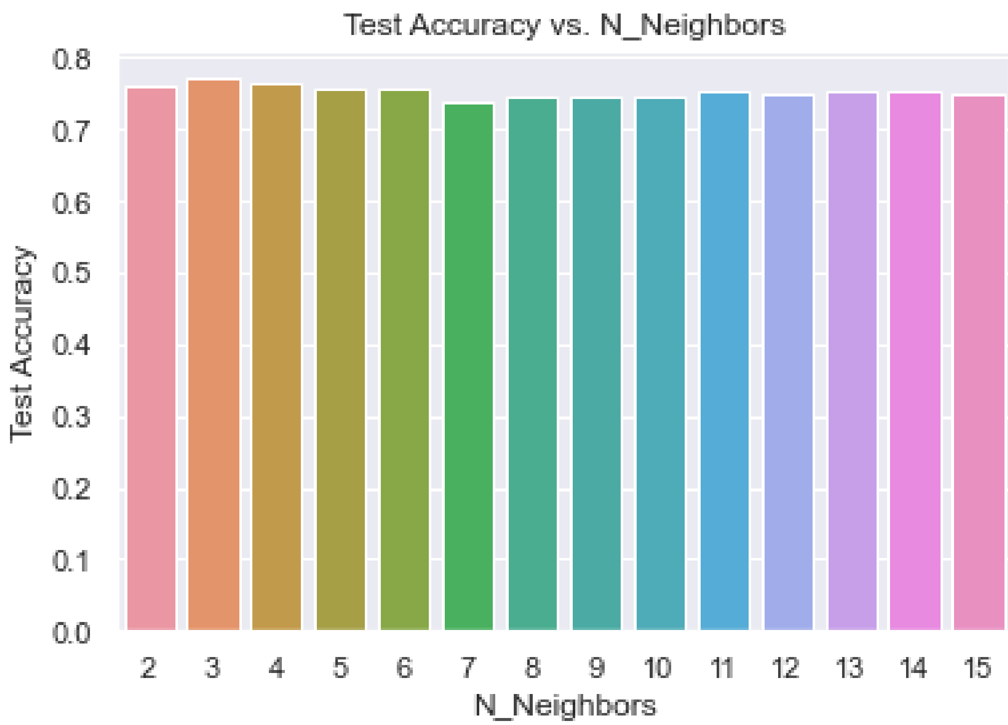
[88]:
```python
gs_knn.best_score_
```

[88]: 0.771

[89]:
```python
# gs_knn.cv_results_
gs_knn.cv_results_['mean_test_score']
```

[89]:
```
array([0.76 , 0.771, 0.765, 0.757, 0.757, 0.739, 0.744, 0.746, 0.744,
       0.755, 0.751, 0.755, 0.754, 0.749])
```

[90]:
```python
plt.figure(figsize=(6,4))
sns.barplot(x=knn_neighbors, y=gs_knn.cv_results_['mean_test_score'])
plt.xlabel("N_Neighbors")
plt.ylabel("Test Accuracy")
plt.title("Test Accuracy vs. N_Neighbors");
```



[91]:
```
knn2 = KNeighborsClassifier(n_neighbors=3)
```

[92]:
```
knn2.fit(X_train, y_train)
```

[92]: KNeighborsClassifier(n_neighbors=3)

[93]:
```
knn2.score(X_train,y_train)
```

[93]: 0.8835294117647059

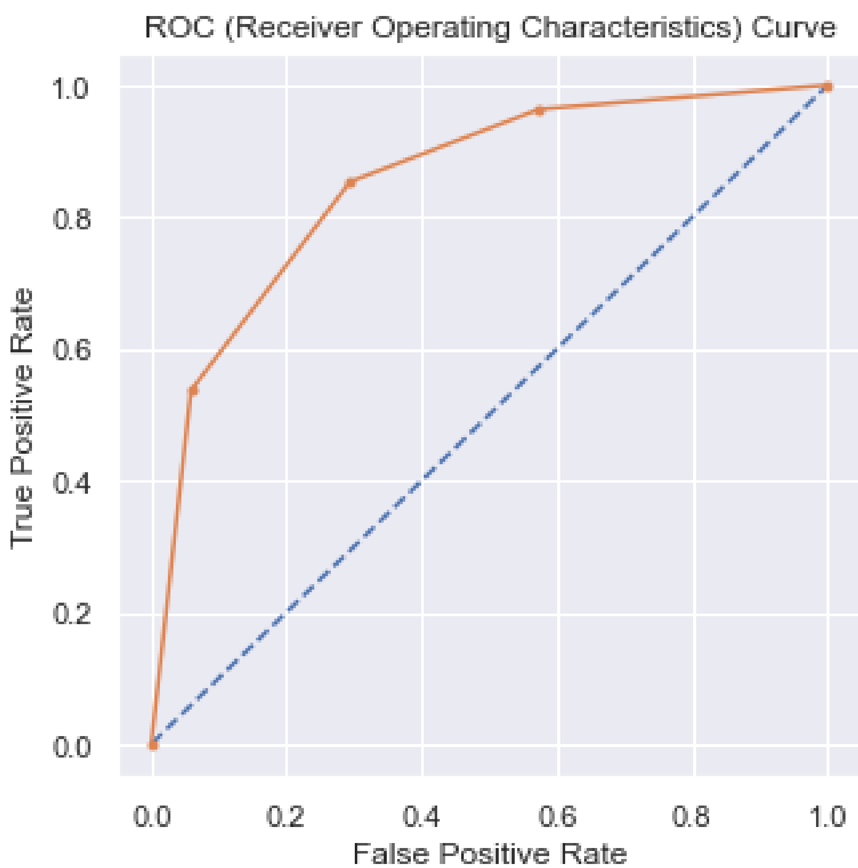
[94]:
```
knn2.score(X_test,y_test)
```

[94]: 0.7866666666666666

[95]:
```
# Preparing ROC Curve (Receiver Operating Characteristics Curve)

probs = knn2.predict_proba(X_test)                # predict probabilities
probs = probs[:, 1]                               # keep probabilities for the␣
↪positive outcome only

auc_knn = roc_auc_score(y_test, probs)            # calculate AUC
print('AUC: %.3f' %auc_knn)
fpr, tpr, thresholds = roc_curve(y_test, probs)  # calculate roc curve
```

```
plt.plot([0, 1], [0, 1], linestyle='--')          # plot no skill
plt.plot(fpr, tpr, marker='.')                    # plot the roc curve for the␣
 ↪model
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC (Receiver Operating Characteristics) Curve");
```

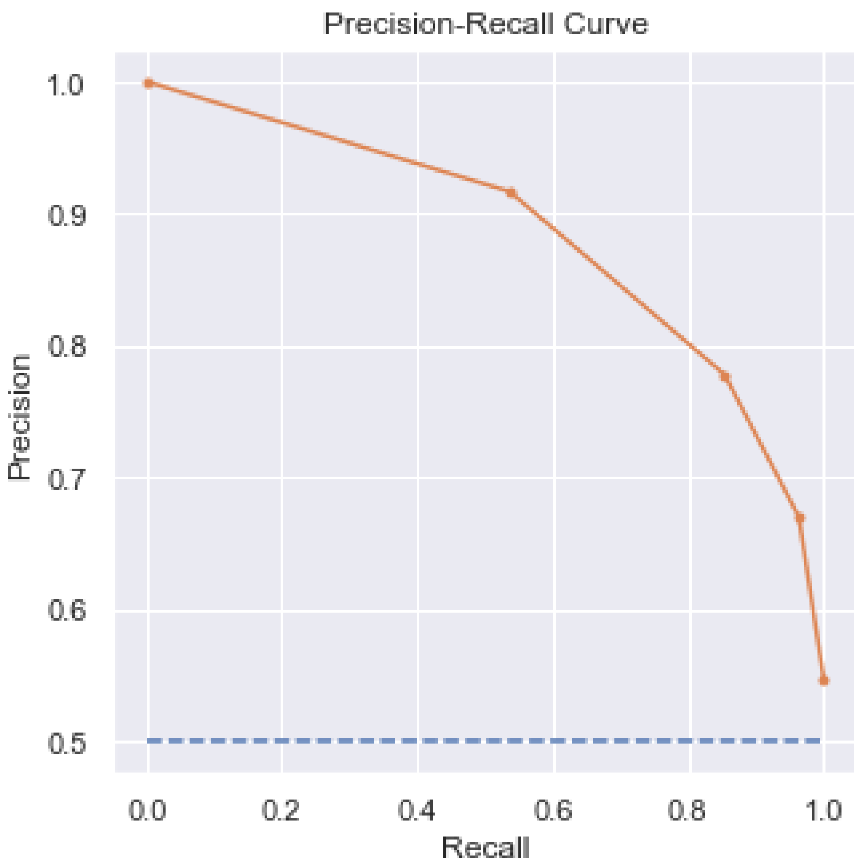
```
AUC: 0.852
```



[96]:
```
# Precision Recall Curve

pred_y_test = knn2.predict(X_test)                                          #␣
 ↪predict class values
precision, recall, thresholds = precision_recall_curve(y_test, probs) #␣
 ↪calculate precision-recall curve
f1 = f1_score(y_test, pred_y_test)                                          #␣
 ↪calculate F1 score
auc_knn_pr = auc(recall, precision)                                         #␣
 ↪calculate precision-recall AUC
```

```
ap = average_precision_score(y_test, probs)                              #␣
 ↪calculate average precision score
print('f1=%.3f auc_pr=%.3f ap=%.3f' % (f1, auc_knn_pr, ap))
plt.plot([0, 1], [0.5, 0.5], linestyle='--')                             # plot no␣
 ↪skill
plt.plot(recall, precision, marker='.')                                  # plot␣
 ↪the precision-recall curve for the model
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve");
```

```
f1=0.814 auc_pr=0.885 ap=0.832
```



[97]:
```
models.append('KNN')
model_accuracy.append(accuracy_score(y_test, pred_y_test))
model_f1.append(f1)
model_auc.append(auc_knn)
```

Support Vector Machine (SVM) Algorithm:

[98]:
```
from sklearn.svm import SVC
svm1 = SVC(kernel='rbf')
```

[99]:
```
svm1.fit(X_train, y_train)
```

[99]:
```
SVC()
```

[100]:
```
svm1.score(X_train, y_train)
```

[100]:
```
0.7282352941176471
```

[101]:
```
svm1.score(X_test, y_test)
```

[101]:
```
0.78
```

Performance evaluation and optimizing parameters using GridSearchCV:

[102]:
```
parameters = {
    'C':[1, 5, 10, 15, 20, 25],
    'gamma':[0.001, 0.005, 0.0001, 0.00001]
}
```

[103]:
```
gs_svm = GridSearchCV(estimator=svm1, param_grid=parameters, cv=5, verbose=0)
gs_svm.fit(df_X_resampled, df_y_resampled)
```

[103]:
```
GridSearchCV(cv=5, estimator=SVC(),
             param_grid={'C': [1, 5, 10, 15, 20, 25],
                         'gamma': [0.001, 0.005, 0.0001, 1e-05]})
```

[104]:
```
gs_svm.best_params_
```

[104]:
```
{'C': 20, 'gamma': 0.005}
```

[105]:
```
gs_svm.best_score_
```

[105]:
```
0.8089999999999999
```

[106]:
```
svm2 = SVC(kernel='rbf', C=20, gamma=0.005, probability=True)
```

[107]:
```
svm2.fit(X_train, y_train)
```

[107]:
```
SVC(C=20, gamma=0.005, probability=True)
```

[108]:
```
svm2.score(X_train, y_train)
```

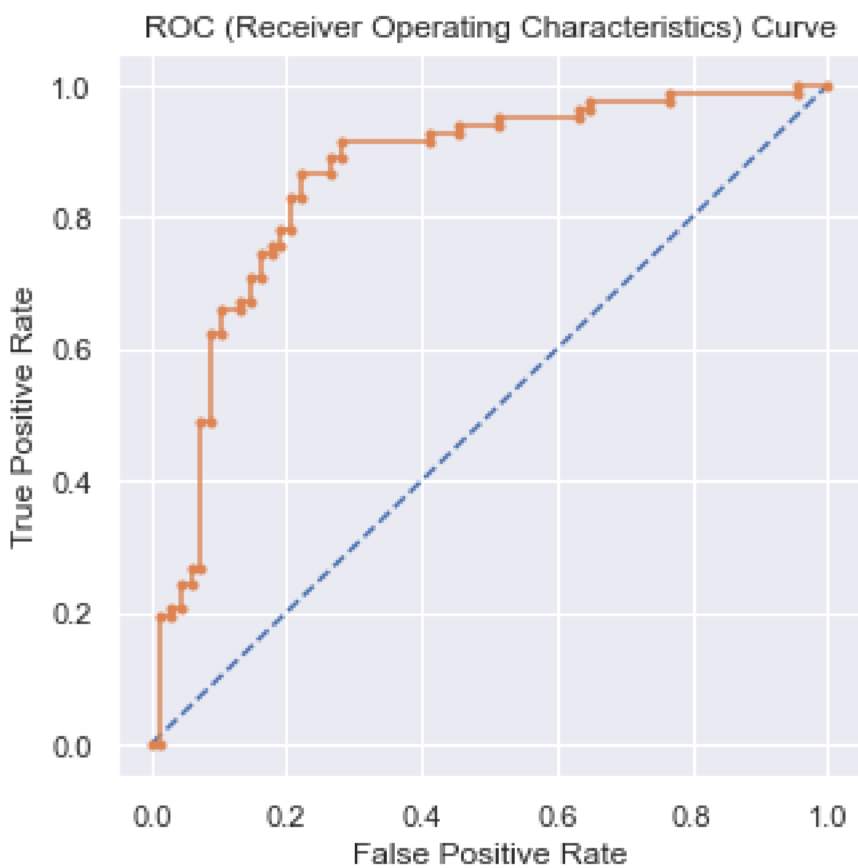
[108]:
```
0.9941176470588236
```

[109]:
```
svm2.score(X_test, y_test)
```

[109]: 0.8133333333333334

[110]:
```
# Preparing ROC Curve (Receiver Operating Characteristics Curve)

probs = svm2.predict_proba(X_test)                  # predict probabilities
probs = probs[:, 1]                                 # keep probabilities for the␣
 ↪positive outcome only

auc_svm = roc_auc_score(y_test, probs)              # calculate AUC
print('AUC: %.3f' %auc_svm)
fpr, tpr, thresholds = roc_curve(y_test, probs)     # calculate roc curve
plt.plot([0, 1], [0, 1], linestyle='--')            # plot no skill
plt.plot(fpr, tpr, marker='.')                      # plot the roc curve for the␣
 ↪model
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC (Receiver Operating Characteristics) Curve");
```

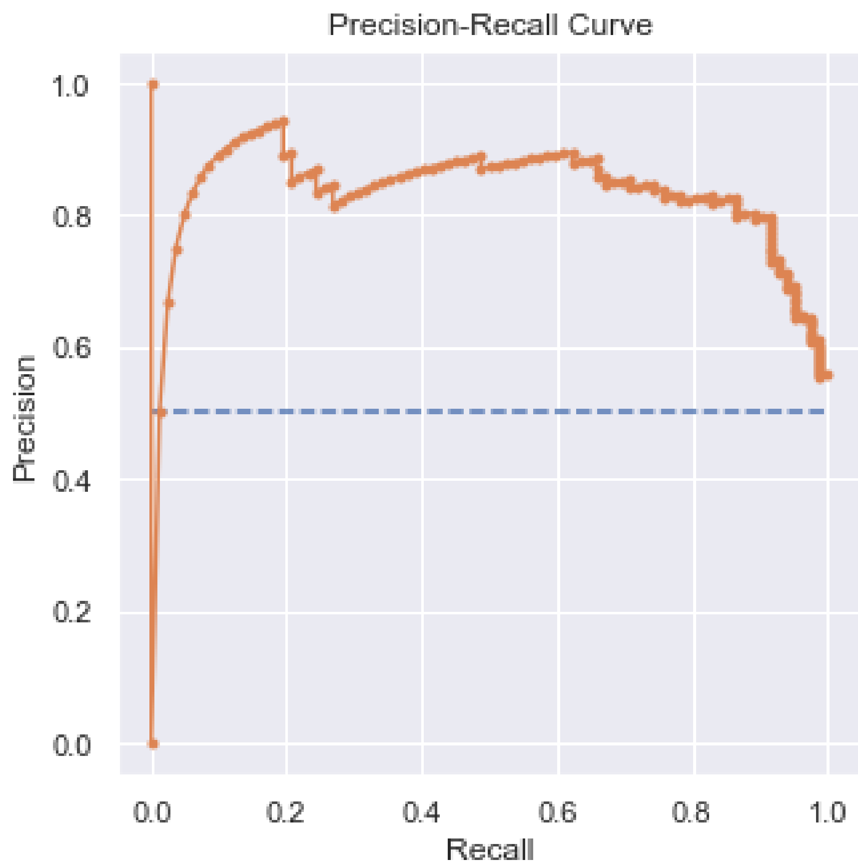
AUC: 0.857



[111]:

```python
# Precision Recall Curve

pred_y_test = svm2.predict(X_test)                                          # predict␣
 ↪class values
precision, recall, thresholds = precision_recall_curve(y_test, probs) #␣
 ↪calculate precision-recall curve
f1 = f1_score(y_test, pred_y_test)                                          #␣
 ↪calculate F1 score
auc_svm_pr = auc(recall, precision)                                         #␣
 ↪calculate precision-recall AUC
ap = average_precision_score(y_test, probs)                                 #␣
 ↪calculate average precision score
print('f1=%.3f auc_pr=%.3f ap=%.3f' % (f1, auc_svm_pr, ap))
plt.plot([0, 1], [0.5, 0.5], linestyle='--')                               # plot no␣
 ↪skill
plt.plot(recall, precision, marker='.')                                     # plot␣
 ↪the precision-recall curve for the model
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve");
```

```
f1=0.829 auc_pr=0.830 ap=0.837
```



[112]:
```
models.append('SVM')
model_accuracy.append(accuracy_score(y_test, pred_y_test))
model_f1.append(f1)
model_auc.append(auc_svm)
```

Naive Bayes Algorithm:

[113]:
```
from sklearn.naive_bayes import GaussianNB, BernoulliNB, MultinomialNB
gnb = GaussianNB()
```

[114]:
```
gnb.fit(X_train, y_train)
```

[114]: GaussianNB()

[115]:
```
gnb.score(X_train, y_train)
```

[115]: 0.7294117647058823

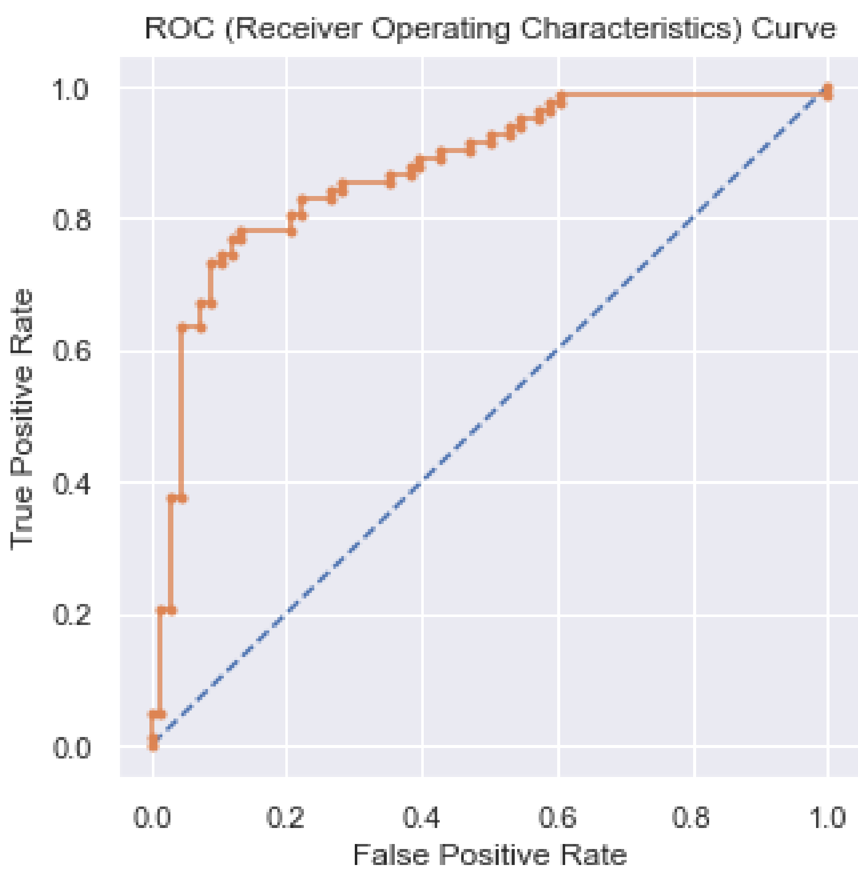
[116]:
```
gnb.score(X_test, y_test)
```

[116]: 0.8

Naive Bayes has almost no hyperparameters to tune, so it usually generalizes well.

[117]:
```
# Preparing ROC Curve (Receiver Operating Characteristics Curve)

probs = gnb.predict_proba(X_test)                   # predict probabilities
probs = probs[:, 1]                                 # keep probabilities for the␣
 ↪positive outcome only

auc_gnb = roc_auc_score(y_test, probs)               # calculate AUC
print('AUC: %.3f' %auc_gnb)
fpr, tpr, thresholds = roc_curve(y_test, probs)     # calculate roc curve
plt.plot([0, 1], [0, 1], linestyle='--')            # plot no skill
plt.plot(fpr, tpr, marker='.')                      # plot the roc curve for the␣
 ↪model
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC (Receiver Operating Characteristics) Curve");
```

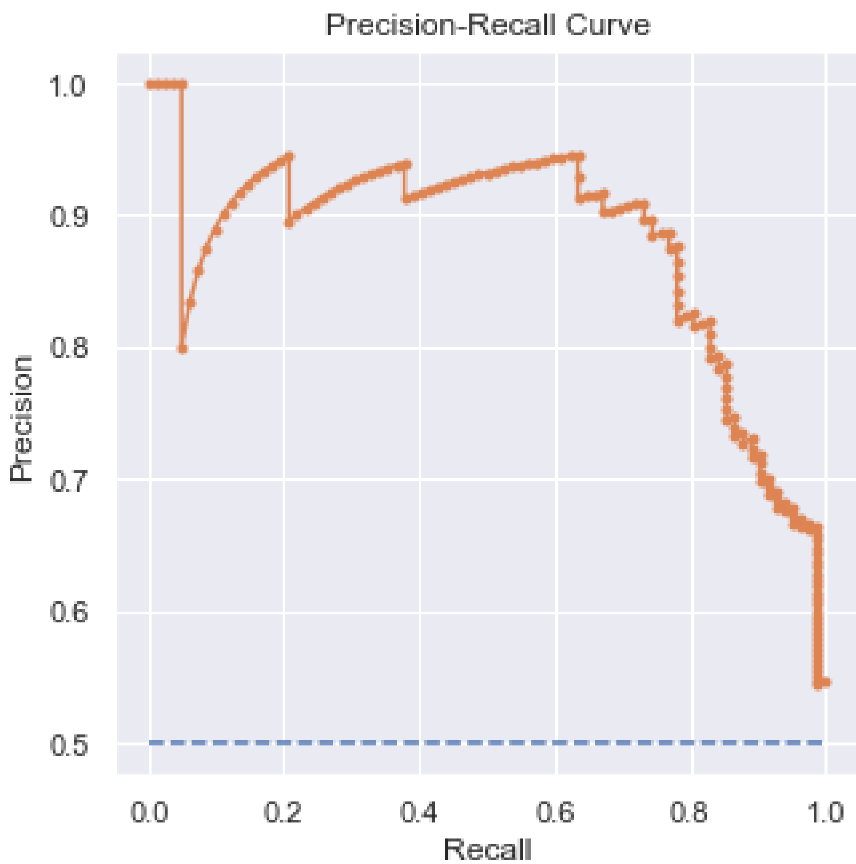
AUC: 0.873



[118]:

```
# Precision Recall Curve

pred_y_test = gnb.predict(X_test)                                         # predict␣
↪class values
precision, recall, thresholds = precision_recall_curve(y_test, probs) #␣
↪calculate precision-recall curve
f1 = f1_score(y_test, pred_y_test)                                        #␣
↪calculate F1 score
auc_gnb_pr = auc(recall, precision)                                        #␣
↪calculate precision-recall AUC
ap = average_precision_score(y_test, probs)                               #␣
↪calculate average precision score
print('f1=%.3f auc_pr=%.3f ap=%.3f' % (f1, auc_gnb_pr, ap))
plt.plot([0, 1], [0.5, 0.5], linestyle='--')                              # plot no␣
↪skill
plt.plot(recall, precision, marker='.')                                   # plot␣
↪the precision-recall curve for the model
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve");
```

```
f1=0.819 auc_pr=0.879 ap=0.880
```



[119]:
```
models.append('GNB')
model_accuracy.append(accuracy_score(y_test, pred_y_test))
model_f1.append(f1)
model_auc.append(auc_gnb)
```

Ensemble Learning –> Boosting –> Adaptive Boosting:

[120]:
```
from sklearn.ensemble import AdaBoostClassifier
ada1 = AdaBoostClassifier(n_estimators=100)
```

[121]:
```
ada1.fit(X_train,y_train)
```

[121]:
```
AdaBoostClassifier(n_estimators=100)
```

[122]:
```
ada1.score(X_train,y_train)
```

[122]:
```
0.8564705882352941
```

[123]:
```
ada1.score(X_test, y_test)
```

[123]: 0.7666666666666667

Performance evaluation and optimizing parameters using cross_val_score:

[124]:
```
parameters = {'n_estimators': [100,200,300,400,500,700,1000]}
```

[125]:
```
gs_ada = GridSearchCV(ada1, param_grid = parameters, cv=5, verbose=0)
gs_ada.fit(df_X_resampled, df_y_resampled)
```

[125]:
```
GridSearchCV(cv=5, estimator=AdaBoostClassifier(n_estimators=100),
             param_grid={'n_estimators': [100, 200, 300, 400, 500, 700, 1000]})
```

[126]:
```
gs_ada.best_params_
```

[126]:
```
{'n_estimators': 500}
```

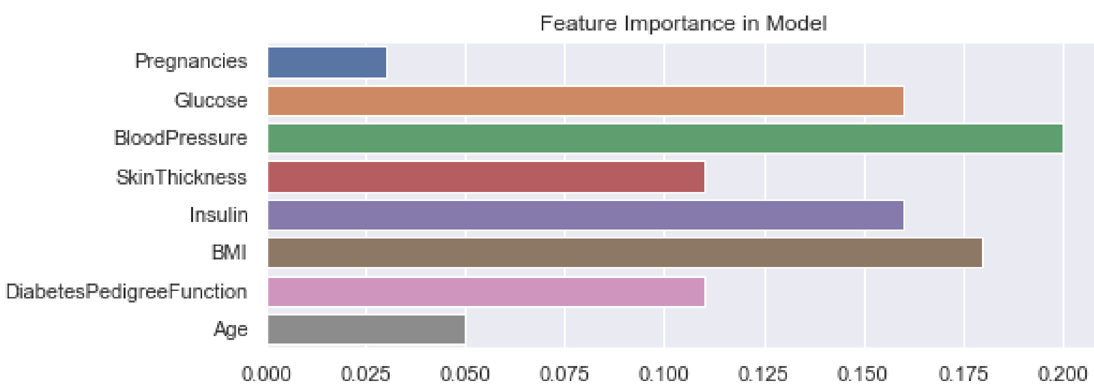
[127]:
```
gs_ada.best_score_
```

[127]: 0.785

[128]:
```
ada1.feature_importances_
```

[128]:
```
array([0.03, 0.16, 0.2 , 0.11, 0.16, 0.18, 0.11, 0.05])
```

[129]:
```
plt.figure(figsize=(8,3))
sns.barplot(y=X_train.columns, x=ada1.feature_importances_)
plt.title("Feature Importance in Model");
```



[130]:
```
ada2 = AdaBoostClassifier(n_estimators=500)
```

[131]:
```
ada2.fit(X_train,y_train)
```

[131]:
```
AdaBoostClassifier(n_estimators=500)
```

[132]:
```
ada2.score(X_train,y_train)
```

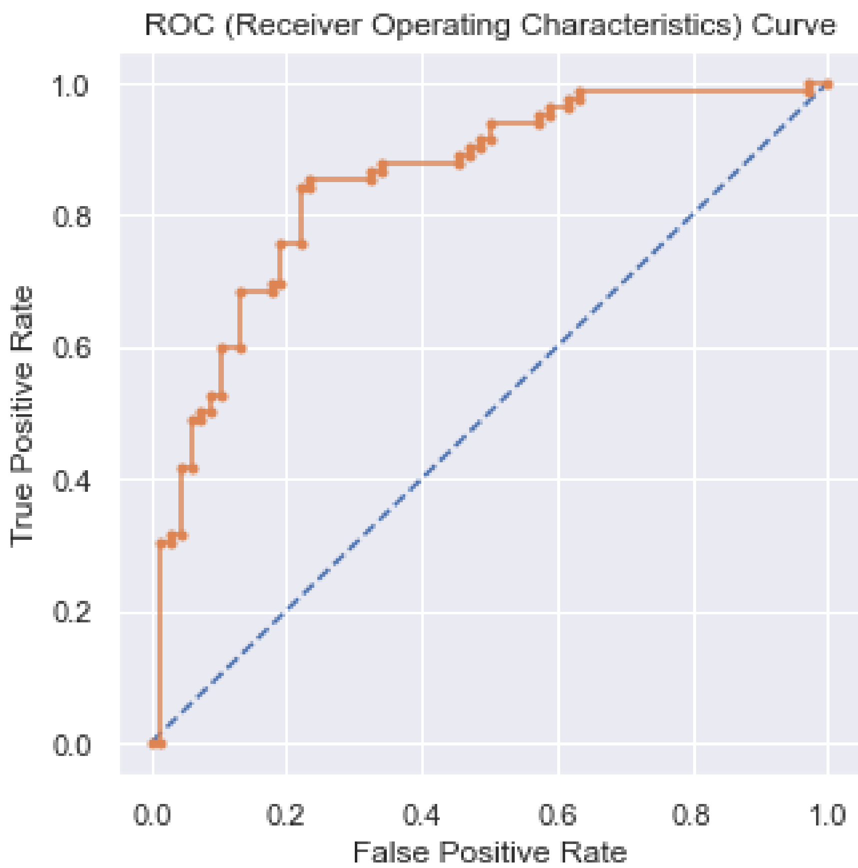
[132]: 0.9247058823529412

[133]:
```
ada2.score(X_test, y_test)
```

[133]: 0.7733333333333333

[134]:
```
# Preparing ROC Curve (Receiver Operating Characteristics Curve)

probs = ada2.predict_proba(X_test)                    # predict probabilities
probs = probs[:, 1]                                   # keep probabilities for the␣
 ↪positive outcome only

auc_ada = roc_auc_score(y_test, probs)                # calculate AUC
print('AUC: %.3f' %auc_ada)
fpr, tpr, thresholds = roc_curve(y_test, probs)  # calculate roc curve
plt.plot([0, 1], [0, 1], linestyle='--')              # plot no skill
plt.plot(fpr, tpr, marker='.')                        # plot the roc curve for the␣
 ↪model
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC (Receiver Operating Characteristics) Curve");
```

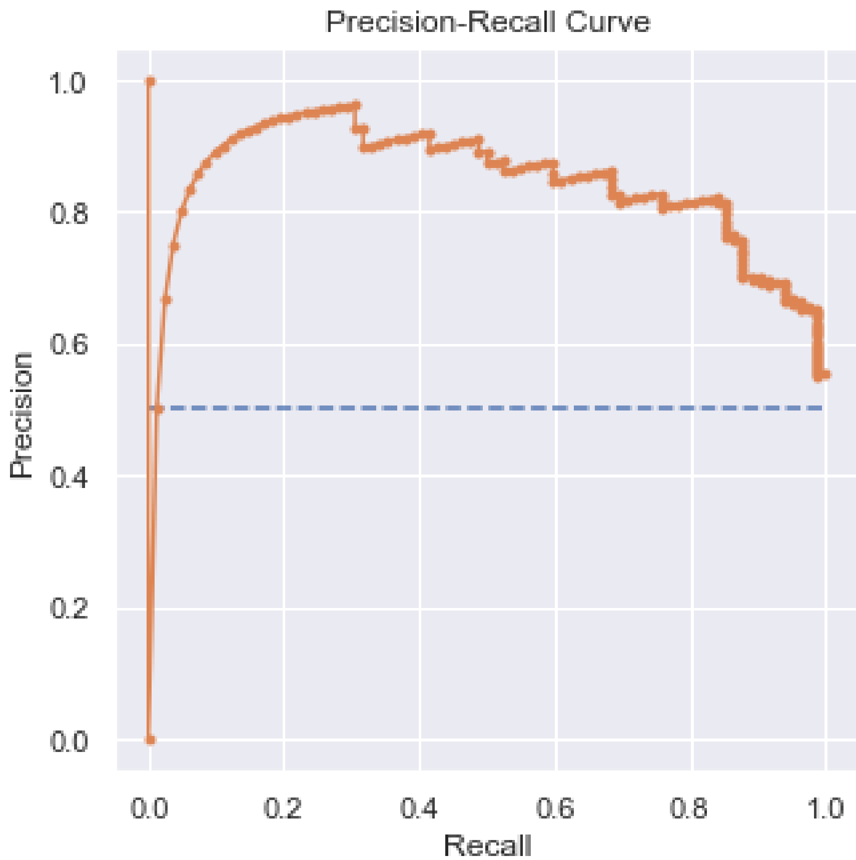
```
AUC: 0.850
```



[135]:
```
# Precision Recall Curve

pred_y_test = ada2.predict(X_test)                                      # predict␣
 ↪class values
precision, recall, thresholds = precision_recall_curve(y_test, probs) #␣
 ↪calculate precision-recall curve
f1 = f1_score(y_test, pred_y_test)                                      #␣
 ↪calculate F1 score
auc_ada_pr = auc(recall, precision)                                     #␣
 ↪calculate precision-recall AUC
ap = average_precision_score(y_test, probs)                             #␣
 ↪calculate average precision score
print('f1=%.3f auc_pr=%.3f ap=%.3f' % (f1, auc_ada_pr, ap))
plt.plot([0, 1], [0.5, 0.5], linestyle='--')                          # plot no␣
 ↪skill
plt.plot(recall, precision, marker='.')                                 # plot␣
 ↪the precision-recall curve for the model
plt.xlabel("Recall")
plt.ylabel("Precision")
```

```
plt.title("Precision-Recall Curve");
```

f1=0.785 auc_pr=0.838 ap=0.845



[136]:
```
models.append('ADA')
model_accuracy.append(accuracy_score(y_test, pred_y_test))
model_f1.append(f1)
model_auc.append(auc_ada)
```

Ensemble Learning –> Boosting –> Gradient Boosting (XGBClassifier):

[137]:
```
from xgboost import XGBClassifier
xgb1 = XGBClassifier(use_label_encoder=False, objective = 'binary:logistic',␣
↪nthread=4, seed=10)
```

[138]:
```
xgb1.fit(X_train, y_train)
```

```
[01:58:23] WARNING: C:/Users/Administrator/workspace/xgboost-
win64_release_1.5.1/src/learner.cc:1115: Starting in XGBoost 1.3.0, the default
evaluation metric used with the objective 'binary:logistic' was changed from
```

```
'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore the
old behavior.
```

[138]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
              gamma=0, gpu_id=-1, importance_type=None,
              interaction_constraints='', learning_rate=0.300000012,
              max_delta_step=0, max_depth=6, min_child_weight=1, missing=nan,
              monotone_constraints='()', n_estimators=100, n_jobs=4, nthread=4,
              num_parallel_tree=1, predictor='auto', random_state=10,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=10,
              subsample=1, tree_method='exact', use_label_encoder=False,
              validate_parameters=1, …)
```

[139]:
```
xgb1.score(X_train, y_train)
```

[139]:
```
1.0
```

[140]:
```
xgb1.score(X_test, y_test)
```

[140]:
```
0.8266666666666667
```

Performance evaluation and optimizing parameters using GridSearchCV:

[141]:
```
parameters = {
    'max_depth': range (2, 10, 1),
    'n_estimators': range(60, 220, 40),
    'learning_rate': [0.1, 0.01, 0.05]
}
```

[142]:
```
gs_xgb = GridSearchCV(xgb1, param_grid = parameters, scoring = 'roc_auc',␣
 ↪n_jobs = 10, cv=5, verbose=0)
gs_xgb.fit(df_X_resampled, df_y_resampled)
```

```
[02:00:05] WARNING: C:/Users/Administrator/workspace/xgboost-
win64_release_1.5.1/src/learner.cc:1115: Starting in XGBoost 1.3.0, the default
evaluation metric used with the objective 'binary:logistic' was changed from
'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore the
old behavior.
```

[142]:
```
GridSearchCV(cv=5,
             estimator=XGBClassifier(base_score=0.5, booster='gbtree',
                                     colsample_bylevel=1, colsample_bynode=1,
                                     colsample_bytree=1,
                                     enable_categorical=False, gamma=0,
                                     gpu_id=-1, importance_type=None,
                                     interaction_constraints='',
                                     learning_rate=0.300000012,
```

```
                                              max_delta_step=0, max_depth=6,
                                              min_child_weight=1, missing=nan,
                                              monotone_constraints='()',
                                              n_estimators=100, n_jobs=4, nthread=4,
                                              num_parallel_tree=1, predictor='auto',
                                              random_state=10, reg_alpha=0, reg_lambda=1,
                                              scale_pos_weight=1, seed=10, subsample=1,
                                              tree_method='exact',
                                              use_label_encoder=False,
                                              validate_parameters=1, …),
                        n_jobs=10,
                        param_grid={'learning_rate': [0.1, 0.01, 0.05],
                                    'max_depth': range(2, 10),
                                    'n_estimators': range(60, 220, 40)},
                        scoring='roc_auc')
```

[143]:
```
gs_xgb.best_params_
```

[143]:
```
{'learning_rate': 0.05, 'max_depth': 7, 'n_estimators': 180}
```

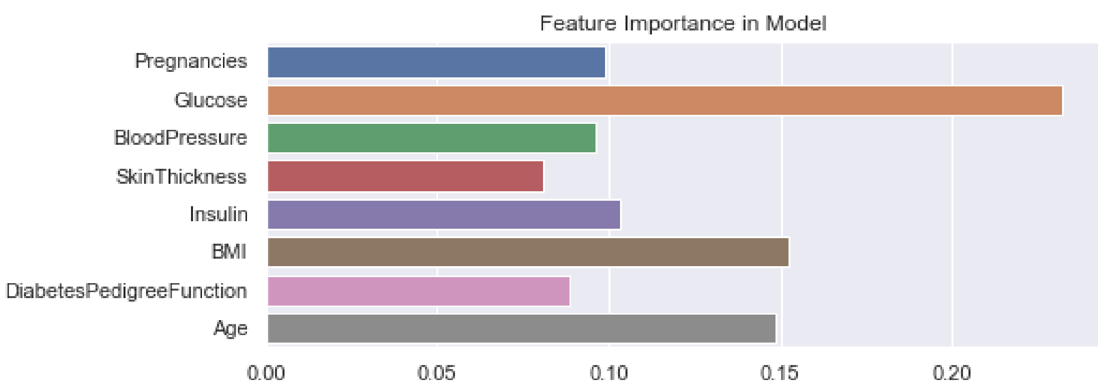
[144]:
```
gs_xgb.best_score_
```

[144]:
```
0.88522
```

[145]:
```
xgb1.feature_importances_
```

[145]:
```
array([0.09883171, 0.23199296, 0.09590795, 0.08073226, 0.10332598,
       0.15247224, 0.08829137, 0.14844562], dtype=float32)
```

[146]:
```
plt.figure(figsize=(8,3))
sns.barplot(y=X_train.columns, x=xgb1.feature_importances_)
plt.title("Feature Importance in Model");
```



[147]:
```
xgb2 = XGBClassifier(use_label_encoder=False, objective = 'binary:logistic',
                    nthread=4, seed=10, learning_rate= 0.05, max_depth= 7,␣
 ↪n_estimators= 180)
```

[148]:
```
xgb2.fit(X_train,y_train)
```

```
[02:00:06] WARNING: C:/Users/Administrator/workspace/xgboost-
win64_release_1.5.1/src/learner.cc:1115: Starting in XGBoost 1.3.0, the default
evaluation metric used with the objective 'binary:logistic' was changed from
'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore the
old behavior.
```

[148]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
              gamma=0, gpu_id=-1, importance_type=None,
              interaction_constraints='', learning_rate=0.05, max_delta_step=0,
              max_depth=7, min_child_weight=1, missing=nan,
              monotone_constraints='()', n_estimators=180, n_jobs=4, nthread=4,
              num_parallel_tree=1, predictor='auto', random_state=10,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=10,
              subsample=1, tree_method='exact', use_label_encoder=False,
              validate_parameters=1, …)
```

[149]:
```
xgb2.score(X_train,y_train)
```

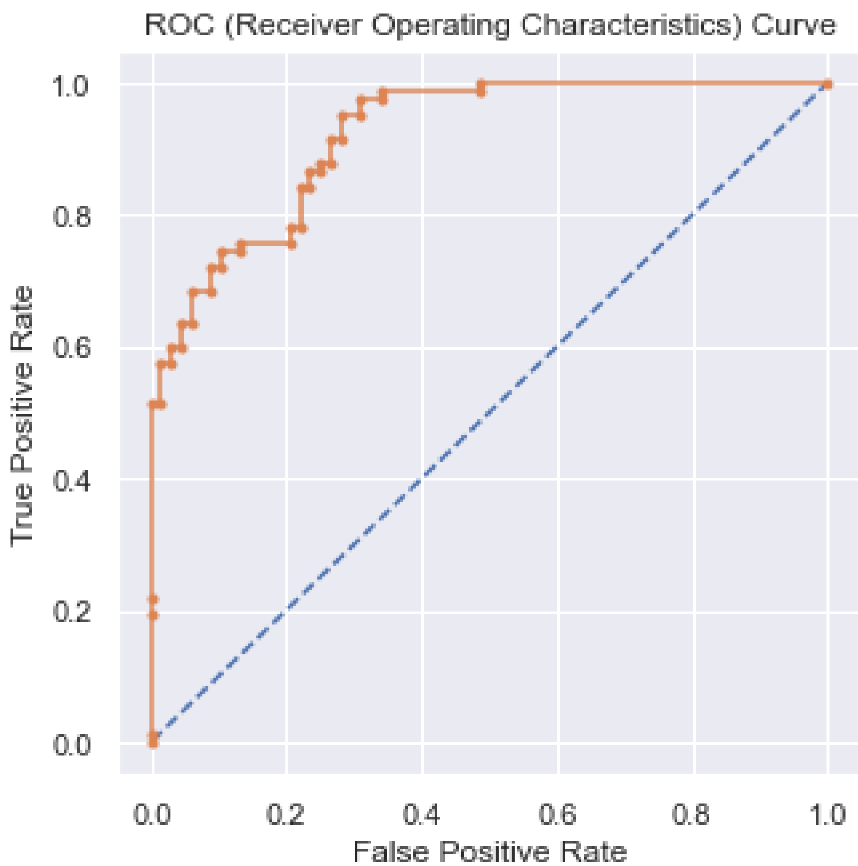
[149]:
```
0.9976470588235294
```

[150]:
```
xgb2.score(X_test, y_test)
```

[150]:
```
0.8066666666666666
```

[151]:
```
# Preparing ROC Curve (Receiver Operating Characteristics Curve)

probs = xgb2.predict_proba(X_test)                  # predict probabilities
probs = probs[:, 1]                                # keep probabilities for the␣
 ↪positive outcome only

auc_xgb = roc_auc_score(y_test, probs)              # calculate AUC
print('AUC: %.3f' %auc_xgb)
fpr, tpr, thresholds = roc_curve(y_test, probs)    # calculate roc curve
plt.plot([0, 1], [0, 1], linestyle='--')           # plot no skill
plt.plot(fpr, tpr, marker='.')                     # plot the roc curve for the␣
 ↪model
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC (Receiver Operating Characteristics) Curve");
```

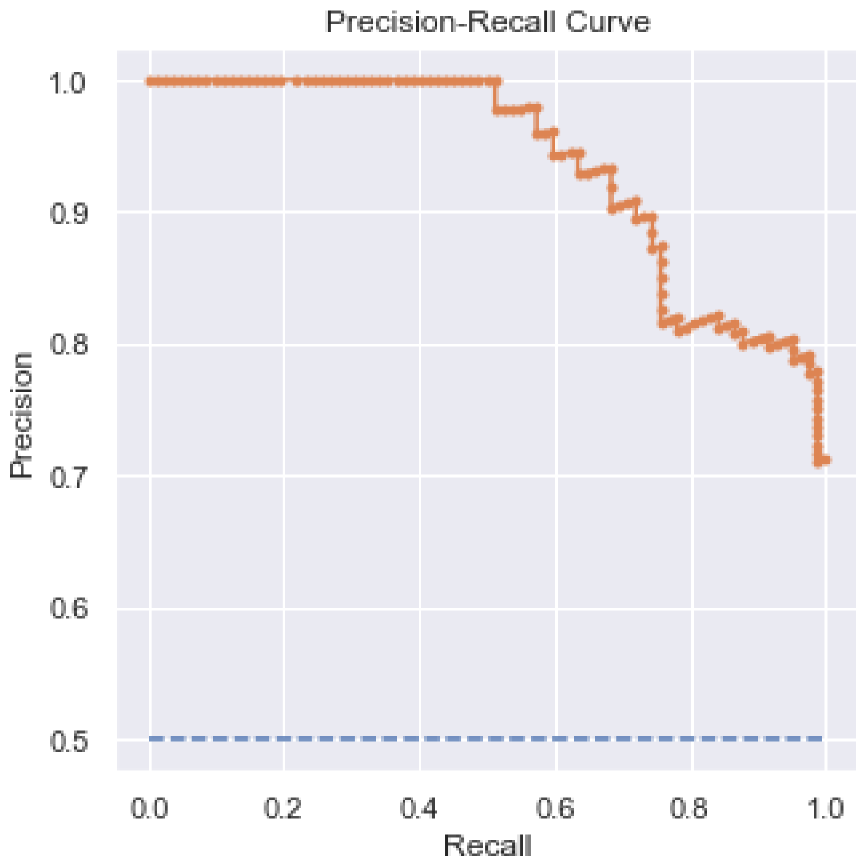
```
AUC: 0.922
```




```
[152]: # Precision Recall Curve

pred_y_test = xgb2.predict(X_test)                                          #␣
 ↪predict class values
precision, recall, thresholds = precision_recall_curve(y_test, probs) #␣
 ↪calculate precision-recall curve
f1 = f1_score(y_test, pred_y_test)                                      #␣
 ↪calculate F1 score
auc_xgb_pr = auc(recall, precision)                                      #␣
 ↪calculate precision-recall AUC
ap = average_precision_score(y_test, probs)                             #␣
 ↪calculate average precision score
print('f1=%.3f auc_pr=%.3f ap=%.3f' % (f1, auc_xgb_pr, ap))
plt.plot([0, 1], [0.5, 0.5], linestyle='--')                            # plot no␣
 ↪skill
plt.plot(recall, precision, marker='.')                                 # plot␣
 ↪the precision-recall curve for the model
plt.xlabel("Recall")
plt.ylabel("Precision")
```

```
plt.title("Precision-Recall Curve");
```

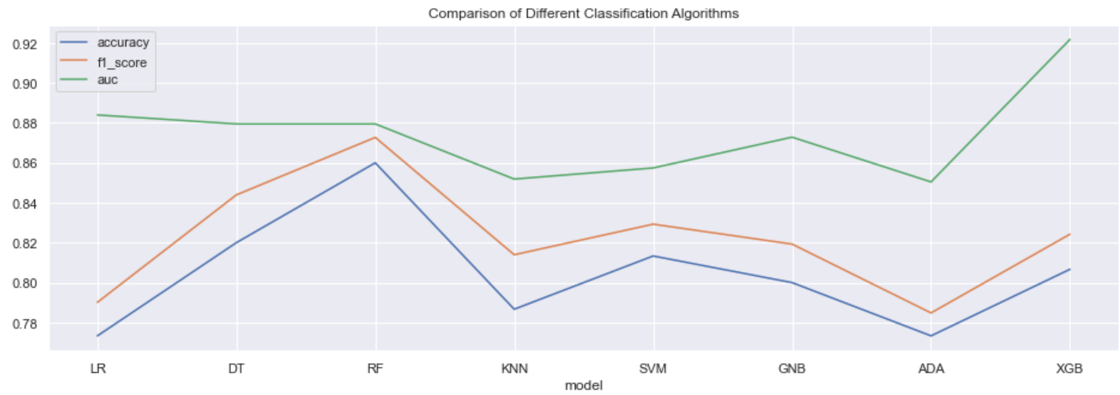
f1=0.824 auc_pr=0.936 ap=0.937



[153]:
```
models.append('XGB')
model_accuracy.append(accuracy_score(y_test, pred_y_test))
model_f1.append(f1)
model_auc.append(auc_xgb)
```

[154]:
```
model_summary = pd.DataFrame(zip(models,model_accuracy,model_f1,model_auc),␣
 ↪columns = ['model','accuracy','f1_score','auc'])
model_summary = model_summary.set_index('model')
```

[155]:
```
model_summary.plot(figsize=(16,5))
plt.title("Comparison of Different Classification Algorithms");
```



[156]:
```
model_summary
```

[156]:

model	accuracy	f1_score	auc
LR	0.773333	0.790123	0.883967
DT	0.820000	0.843931	0.879484
RF	0.860000	0.872727	0.879484
KNN	0.786667	0.813953	0.851865
SVM	0.813333	0.829268	0.857425
GNB	0.800000	0.819277	0.872848
ADA	0.773333	0.784810	0.850430
XGB	0.806667	0.824242	0.921808

Among all models, RandomForest has given best accuracy and f1_score. Therefore we will build final model using RandomForest.

FINAL CLASSIFIER:

[157]:
```
final_model = rf2
```

[158]:
```
cr = classification_report(y_test, final_model.predict(X_test))
print(cr)
```

```
              precision    recall  f1-score   support

           0       0.85      0.84      0.84        68
           1       0.87      0.88      0.87        82

    accuracy                           0.86       150
   macro avg       0.86      0.86      0.86       150
weighted avg       0.86      0.86      0.86       150
```

[159]:
```
confusion = confusion_matrix(y_test, final_model.predict(X_test))
print("Confusion Matrix:\n", confusion)
```

```
Confusion Matrix:
 [[57 11]
 [10 72]]
```

[160]:
```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

Accuracy = (TP+TN)/(TP+TN+FP+FN)
Precision = TP/(TP+FP)
Sensitivity = TP/(TP+FN)                     # also called recall
Specificity = TN/(TN+FP)
```

[161]:
```
print("Accuracy: %.3f"%Accuracy)
print("Precision: %.3f"%Precision)
print("Sensitivity: %.3f"%Sensitivity)
print("Specificity: %.3f"%Specificity)
print("AUC: %.3f"%auc_rf)
```

```
Accuracy: 0.860
Precision: 0.867
Sensitivity: 0.878
Specificity: 0.838
AUC: 0.928
```