Predicting Presence of Malignant and Benign Tumors given their characteristics using KNN

```
df = pd.read_csv(r"C:\Users\core i5\Documents\GitHub\DataScience\
datascience\CPE 312\KNN-SVM-NaiveBayes\knn.csv", header = None)
df.shape
```

```
(699, 11)
```

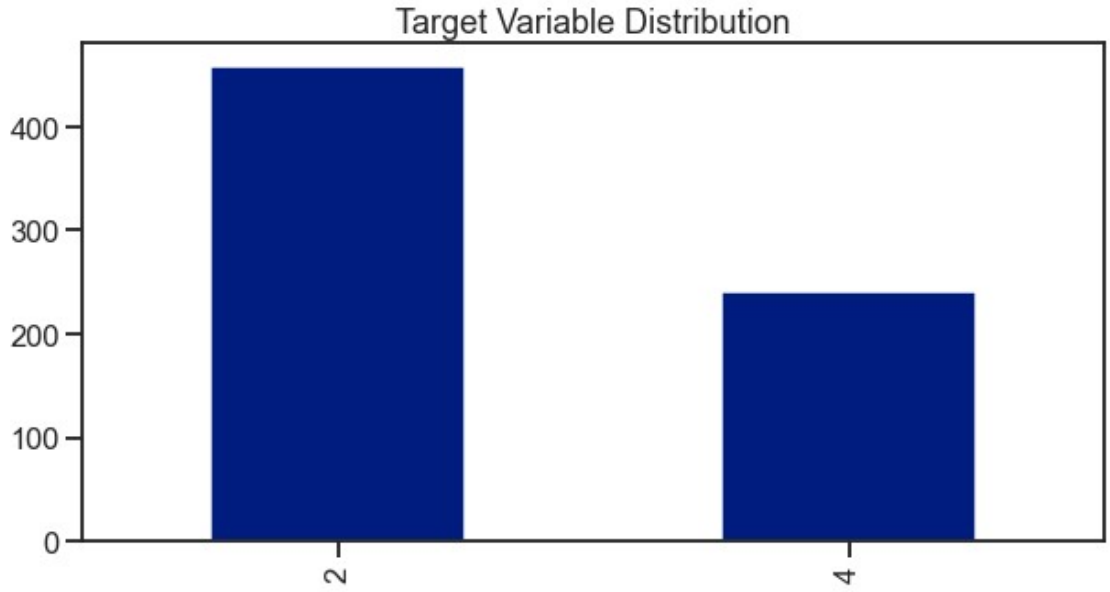
```
# add our columns
col_names = ['Id', 'Clump_thickness', 'Uniformity_Cell_Size',
'Uniformity_Cell_Shape', 'Marginal_Adhesion',
'Single_Epithelial_Cell_Size', 'Bare_Nuclei', 'Bland_Chromatin',
'Normal_Nucleoli', 'Mitoses', 'Class']

df.columns = col_names
df.head()
```

```
        Id  Clump_thickness  Uniformity_Cell_Size
Uniformity_Cell_Shape  \
0  1000025                5                     1
1
1  1002945                5                     4
4
2  1015425                3                     1
1
3  1016277                6                     8
8
4  1017023                4                     1
1

   Marginal_Adhesion  Single_Epithelial_Cell_Size Bare_Nuclei  \
0                  1                            2           1
1                  5                            7          10
2                  1                            2           2
3                  1                            3           4
4                  3                            2           1

   Bland_Chromatin  Normal_Nucleoli  Mitoses  Class
0                3                1        1      2
1                3                2        1      2
2                3                1        1      2
3                3                7        1      2
4                3                1        1      2
```

```
#determine the types of target variables and their
distributions/frequency
```

```
df.iloc[:,-1].value_counts().plot(kind='bar', title='Target Variable
Distribution', figsize=(10,5))
```

```
<AxesSubplot:title={'center':'Target Variable Distribution'}>
```




```
df.drop('Id', axis=1, inplace=True)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 699 entries, 0 to 698
Data columns (total 10 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   Clump_thickness              699 non-null    int64
 1   Uniformity_Cell_Size         699 non-null    int64
 2   Uniformity_Cell_Shape        699 non-null    int64
 3   Marginal_Adhesion            699 non-null    int64
 4   Single_Epithelial_Cell_Size  699 non-null    int64
 5   Bare_Nuclei                  699 non-null    object
 6   Bland_Chromatin              699 non-null    int64
 7   Normal_Nucleoli              699 non-null    int64
 8   Mitoses                      699 non-null    int64
 9   Class                        699 non-null    int64
dtypes: int64(9), object(1)
memory usage: 54.7+ KB
```

```
df['Bare_Nuclei'] = pd.to_numeric(df['Bare_Nuclei'], errors='coerce')
```

```
df.info()
```

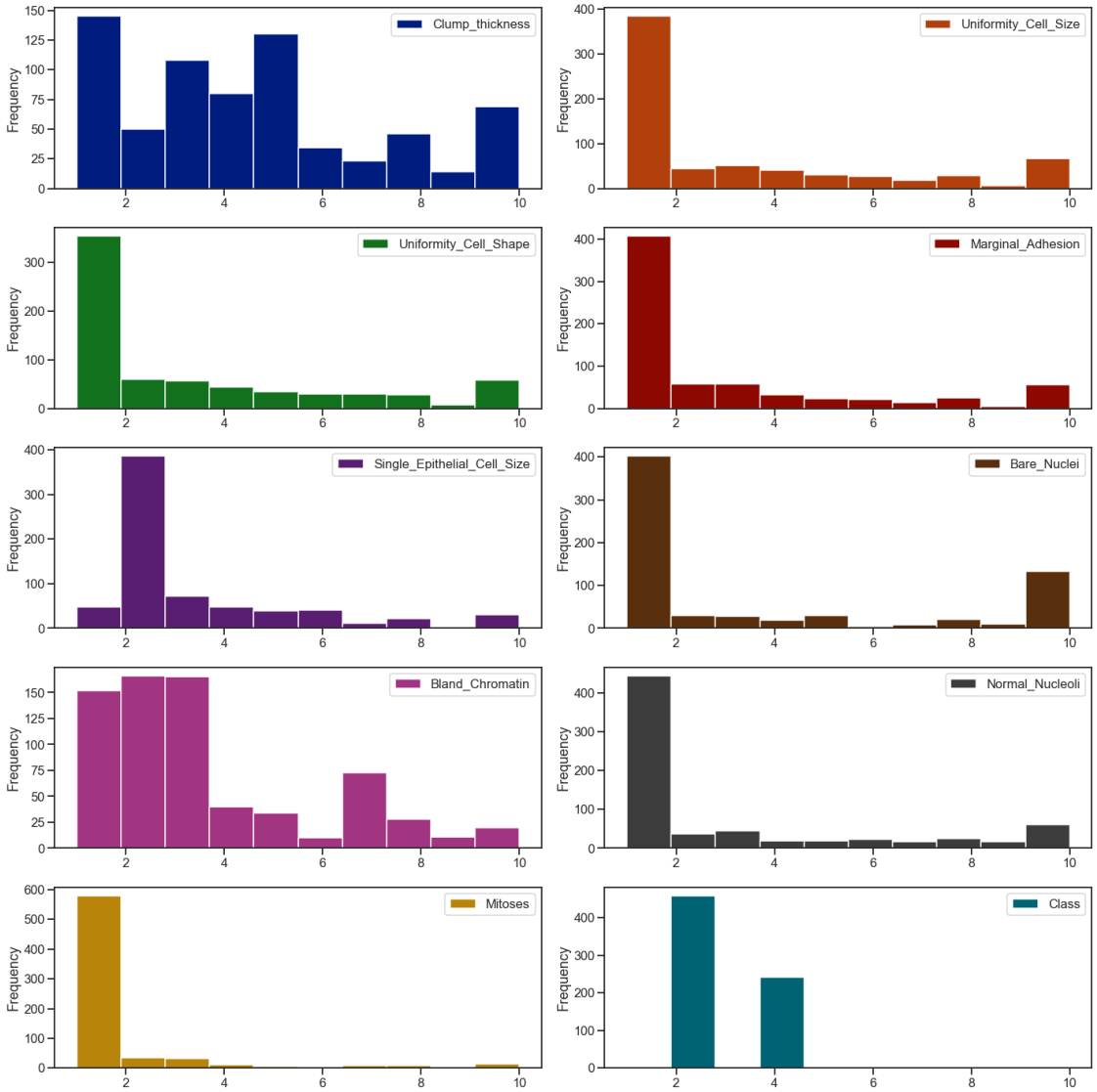
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 699 entries, 0 to 698
```

```
Data columns (total 10 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   Clump_thickness              699 non-null    int64
 1   Uniformity_Cell_Size         699 non-null    int64
 2   Uniformity_Cell_Shape        699 non-null    int64
 3   Marginal_Adhesion            699 non-null    int64
 4   Single_Epithelial_Cell_Size  699 non-null    int64
 5   Bare_Nuclei                  683 non-null    float64
 6   Bland_Chromatin              699 non-null    int64
 7   Normal_Nucleoli              699 non-null    int64
 8   Mitoses                      699 non-null    int64
 9   Class                        699 non-null    int64
dtypes: float64(1), int64(9)
memory usage: 54.7 KB
```

```
# check for missing values
df.isnull().sum()
```

```
Clump_thickness                0
Uniformity_Cell_Size           0
Uniformity_Cell_Shape          0
Marginal_Adhesion              0
Single_Epithelial_Cell_Size    0
Bare_Nuclei                   16
Bland_Chromatin                0
Normal_Nucleoli                0
Mitoses                        0
Class                          0
dtype: int64
```

```
df.plot(kind = "hist", bins = 10, subplots= True, layout = (5,2),
sharex = False, sharey = False, figsize = (20,20))
plt.tight_layout()
plt.show()
```



Feature columns are positively/right skewed.

```
df.corr().iloc[:,-1].sort_values(ascending=False)
```

```
Class                          1.000000
Bare_Nuclei                    0.822696
Uniformity_Cell_Shape          0.818934
Uniformity_Cell_Size           0.817904
Bland_Chromatin                0.756616
Clump_thickness                0.716001
Normal_Nucleoli                0.712244
Marginal_Adhesion              0.696800
Single_Epithelial_Cell_Size    0.682785
Mitoses                        0.423170
Name: Class, dtype: float64
```

all of our feature values are positively correlated with class.

Splitting the dataset into training, validation, and test sets

```
X = df.drop('Class', axis=1)
y = df['Class']

train_val_df , test_df = train_test_split(df, test_size=0.2,
random_state=101)
train_df, val_df = train_test_split(train_val_df, test_size=0.25,
random_state=101)

print("shape of train_df is {}".format(train_df.shape))
print("shape of validity_df is {}".format(val_df.shape))
print("shape of test_df is {}".format(test_df.shape))
```

```
shape of train_df is (419, 10)
shape of validity_df is (140, 10)
shape of test_df is (140, 10)
```

```
train_inputs = train_df.drop('Class', axis=1).copy()
train_target = train_df["Class"].copy()
val_inputs = val_df.drop('Class', axis=1).copy()
val_target = val_df["Class"].copy()
test_inputs = test_df.drop('Class', axis=1).copy()
test_target = test_df["Class"].copy()
```

Imputing missing data

```
# check the columns that have nan values and how many
train_inputs.isnull().sum()
```

```
Clump_thickness               0
Uniformity_Cell_Size          0
Uniformity_Cell_Shape         0
Marginal_Adhesion             0
Single_Epithelial_Cell_Size   0
Bare_Nuclei                   8
Bland_Chromatin               0
Normal_Nucleoli               0
Mitoses                       0
dtype: int64
```

```
# check if data in Bare_Nuclei have outliers. If there are many we can
use median instead of mean.
print(val_inputs.Bare_Nuclei.mean(),
val_inputs.Bare_Nuclei.median())
print(train_inputs.Bare_Nuclei.mean(),
train_inputs.Bare_Nuclei.median())
print(test_inputs.Bare_Nuclei.mean(),
test_inputs.Bare_Nuclei.median())
```

```
3.3582089552238807 1.0
3.506082725060827 1.0
3.8405797101449277 1.0
```

Since the mean and media are different, we can infer that the distribution is skewed. Actually, as we saw earlier all of the data are positvely skewed. I think it is best that we pick the median number to fill out missing valuse for the column `Bare_Nuclei`.

```
from sklearn.impute import SimpleImputer

#create an imputer object
imputer = SimpleImputer(strategy="median")

# fit the imputer model to fill each column with missing values the mean value for that column
imputer.fit(train_inputs)
```

```
SimpleImputer(strategy='median')
```

```
# the object imputer now contains an atribute called .statistics_ which contains the mean value for each column. We can access this:
list(imputer.statistics_)
```

```
[4.0, 1.0, 1.0, 1.0, 2.0, 1.0, 3.0, 1.0, 1.0]
```

```
num_cols =
train_inputs.select_dtypes(include=np.number).columns.tolist()
num_cols
```

```
['Clump_thickness',
 'Uniformity_Cell_Size',
 'Uniformity_Cell_Shape',
 'Marginal_Adhesion',
 'Single_Epithelial_Cell_Size',
 'Bare_Nuclei',
 'Bland_Chromatin',
 'Normal_Nucleoli',
 'Mitoses']
```

```
# we need to inject these values in the predictor variable for all our datasets.

train_inputs[num_cols] = imputer.fit_transform(train_inputs[num_cols])
val_inputs[num_cols] = imputer.fit_transform(val_inputs[num_cols])
test_inputs[num_cols] = imputer.fit_transform(test_inputs[num_cols])

# check to see if there are any null/na values in our data. test data is used here but all datasets have been checked for missing values and all columns are filled.
test_inputs.isnull().sum()
```

```
Clump_thickness               0
Uniformity_Cell_Size          0
Uniformity_Cell_Shape         0
Marginal_Adhesion             0
```

```
Single_Epithelial_Cell_Size    0
Bare_Nuclei                    0
Bland_Chromatin                0
Normal_Nucleoli                0
Mitoses                        0
dtype: int64
```

Normalizing our Numerical columns

```
from sklearn.preprocessing import MinMaxScaler

#create an object for MinMaxScaler
scaler = MinMaxScaler()

#scaler.transform({data fram with num cols}) will result in the
scaling of the values from (0,1)

train_inputs[num_cols] = scaler.fit_transform(train_inputs[num_cols])
val_inputs[num_cols] = scaler.fit_transform(val_inputs[num_cols])
test_inputs[num_cols] = scaler.fit_transform(test_inputs[num_cols])

# verify that the scaling worked (val_inputs is used to see the max
and min but train_inputs and test_inputs both have a max of 1 and min
of 0)
val_inputs[num_cols].describe().loc[["min","max"]]
```

```
     Clump_thickness  Uniformity_Cell_Size  Uniformity_Cell_Shape  \
min              0.0                   0.0                    0.0
max              1.0                   1.0                    1.0

     Marginal_Adhesion  Single_Epithelial_Cell_Size  Bare_Nuclei  \
min                0.0                          0.0          0.0
max                1.0                          1.0          1.0

     Bland_Chromatin  Normal_Nucleoli  Mitoses
min              0.0              0.0      0.0
max              1.0              1.0      1.0
```

Changin the target values from 2->0 and 4->1

0 : benign cancer 1 : malignant cancer

```
val_target.replace({2:0, 4:1}, inplace=True)

train_target.replace({2:0, 4:1}, inplace=True)
test_target.replace({2:0, 4:1}, inplace=True)

# all target values have been change, only train is shown here.
train_target.value_counts()
```

```
0    274
1    145
Name: Class, dtype: int64
```

Training KNN Model

base model

```
from sklearn.neighbors import KNeighborsClassifier

knn_model = KNeighborsClassifier(n_jobs=-1)
knn_model.fit(train_inputs, train_target)
```

```
KNeighborsClassifier(n_jobs=-1)
```

```
knn_model.get_params()
```

```
{'algorithm': 'auto',
 'leaf_size': 30,
 'metric': 'minkowski',
 'metric_params': None,
 'n_jobs': -1,
 'n_neighbors': 5,
 'p': 2,
 'weights': 'uniform'}
```

```
val_pred = knn_model.predict(val_inputs)
val_pred = pd.Series(val_pred)
val_pred.value_counts()
```

```
0    93
1    47
dtype: int64
```

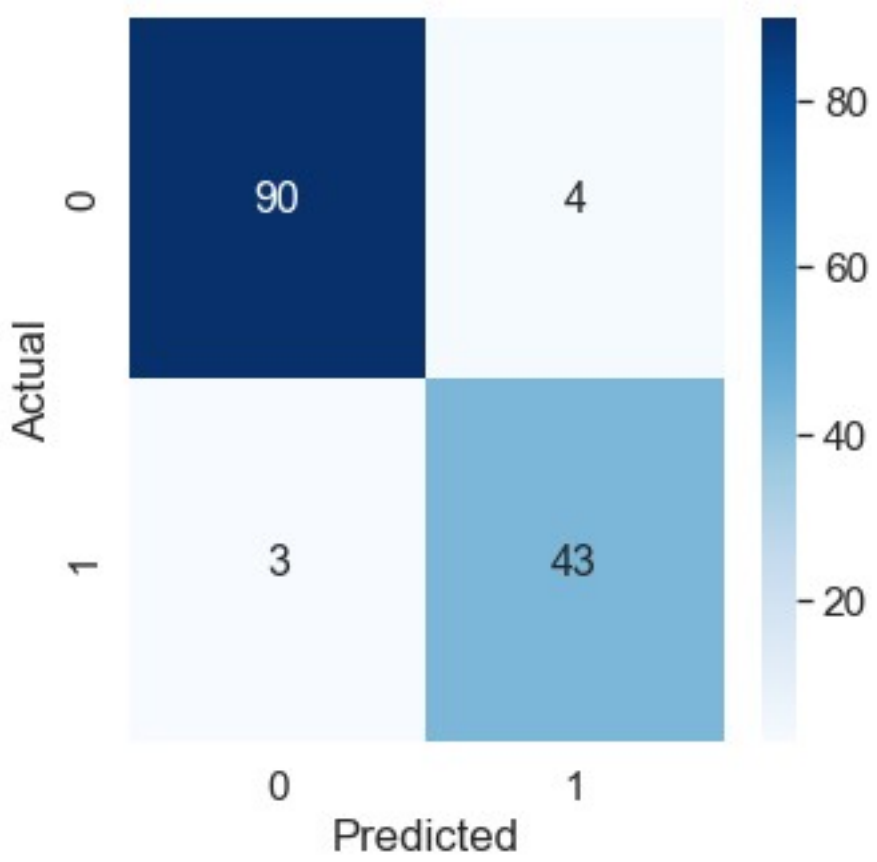
Classification Metrics prior to Hypertuning

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
import matplotlib.pyplot as plt

def pre_recall_curve(clf, input, target):
    '''
        target: target column
        input: input columns
        clf: classifier

    #? Function Outputs a precision(y-axis) vs recall/senstivity(x-
axis) graph. It also shows the F1 score as well as the area under the
curve (AUC) score which summarizes the performance of the model.
```

```
    #! Currently configure to work for binary target columns/data
    #! Best metric for imbalanced datasets
    #* compare the AUC scores for pre_recall_curve VS. roc_curve and
notice if there is difference.
    #* Key feature is that True Negatives (or the class that is the
majority) are not taken into consideration. It is only concerned with
the prediction of the minority class.
    #* This helps interpret the performance of models with imbalanced
datasets because the difference between the two curves is between
Precision (TP/(TP+FN)) and 1-Specificity/FPR (1-(TN/(TN+FP)))

        Given: TN = 1310; TP = 90; FP = 116
        FPR/1-Specificity/X-axis/ROC_curve = 0.08 (ideal, should be
close to 0)
        Precision/Y-axis/Precision_Recall_curve = 0.43 (not ideal,
should be close to 1)

    #! Looking at the ROC curve we might be misled that the score of
0.08 is ideal when in fact this is not ideal at all since the formula
to get this utilized the majority class, and therefore a big numerator
value. This will cause the evaluator to neglect the TP values and FP
values (which we want to increase and decrease, respectively), thus
not allowing our model to be modified to better predict the positive
class. You will end up with a class that is not good with predicting
1s.
    #! We neglected to evaluate the difference between ROC-curve's
sensitivity and pr_recall_curve's recall since they are the same
formula and produce the same values.
    '''
    from sklearn.metrics import precision_recall_curve
    from sklearn.metrics import f1_score
    from matplotlib import pyplot

    lr_precision, lr_recall, _ = precision_recall_curve(target,
clf.predict_proba(input)[:,1], pos_label =
target.value_counts().sort_values(ascending=False).index[-1])

    lr_f1, lr_auc = f1_score(target, clf.predict(input), pos_label =
target.value_counts().sort_values(ascending=False).index[-1]),
auc(lr_recall, lr_precision)
    # summarize scores
    print('KNN: f1=%.3f auc=%.3f' % (lr_f1, lr_auc))
    # plot the precision-recall curves
    no_skill = len(target[target==
target.value_counts().sort_values(ascending=False).index[-1]]) /
len(target)
    pyplot.plot([0, 1], [no_skill, no_skill], linestyle='--',
label='No Skill/baseline: {0:.2f}'.format(no_skill))
    pyplot.plot(lr_recall, lr_precision, marker='.', label='KNN')
    # axis labels
```

```
    pyplot.xlabel('Recall')
    pyplot.ylabel('Precision')
    # show the legend
    pyplot.legend()
    pyplot.plot(1.,1., marker='.')
    pyplot.annotate(xy=[1.,1.],xytext=[0.8,1.05], text="optimal point
(1,1)", size = 10)
    # show the plot
    pyplot.tight_layout()
    pyplot.show()

def accuracyscores(clf, prior_inputs, prior_target, new_inputs ,
new_target):
    print("train model score {}".format(clf.score(prior_inputs,
prior_target)))
    print("non-train model score {}".format(clf.score(new_inputs,
new_target)))

    y_majority = np.full(new_target.shape[0],
new_target.value_counts().sort_values(ascending=False).index[0])
    y_random =
np.random.choice(val_target.value_counts().sort_values(ascending=False
).index.tolist(), new_target.shape[0])

    print("random model accuracy score
{}".format(accuracy_score(new_target, y_random)))
    print("majority model accuracy score
{}".format(accuracy_score(new_target, y_majority)))

def confusionmatrixplot(clf,inputs, target):
    prediction = clf.predict(inputs)

    data = confusion_matrix(target, prediction)
    df_cm = pd.DataFrame(data, columns=np.unique(prediction), index =
np.unique(target))
    df_cm.index.name = 'Actual'
    df_cm.columns.name = 'Predicted'
    plt.figure(figsize = (5,5))
    sns.set(font_scale=1.4)#for label size
    sns.heatmap(df_cm, cmap="Blues", fmt=
'd',annot=True,annot_kws={"size": 16});

def plot_multiclass_roc(clf, X_test, y_test, n_classes, figsize=(17,
6)):
    try:
        y_score = clf.decision_function(X_test)
        print("Using decision_function method")
    except:
        y_score = clf.predict_proba(X_test)
        print("Using predict_proba method")
```

```
    # structures
    fpr = dict()
    tpr = dict()
    roc_auc = dict()

    # calculate dummies once
    y_test_dummies = pd.get_dummies(y_test, drop_first=False).values
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_test_dummies[:, i], y_score[:,
i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # roc for each class
    fig, ax = plt.subplots(figsize=figsize)
    ax.plot([0, 1], [0, 1], 'k--')
    ax.set_xlim([0.0, 1.0])
    ax.set_ylim([0.0, 1.05])
    ax.set_xlabel('False Positive Rate')
    ax.set_ylabel('True Positive Rate')
    ax.set_title('Receiver operating characteristic for
RandomForest_roc_auc_cruve')
    for i in range(n_classes):
        ax.plot(fpr[i], tpr[i], label='ROC curve (area = %0.2f) for
label %i' % (roc_auc[i], i))
    ax.legend(loc="best")
    ax.grid(alpha=.4)
    sns.despine()
    plt.tight_layout()
    plt.show();

accuracyscores(knn_model, train_inputs, train_target, val_inputs,
val_target)

train model score 0.9737470167064439
non-train model score 0.95
random model accuracy score 0.5071428571428571
majority model accuracy score 0.6714285714285714

There does not seem to be any case of overfitting since our model was
able to generalize on our validation dataset. The scores for the
training and validation datasets are high and comparable.

We can benchmark our model against baseline models wherein the
predicted values are either random (y_random) or contains the majority
target class (y_majority). Checking their accuracies and comparing our
model's own accuracy against these dumb models can give as a pretty
good idea whether our model is worthwhile and better. Fortunately, our
model outperformed both the dumb models using the accuracy metric.

confusionmatrixplot(knn_model, val_inputs, val_target)
```



So far our model is looking very promising! There are very few False Negatives and False Positives in our predictions.

```
print(classification_report(val_target,
knn_model.predict(val_inputs)))
```

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96        94
           1       0.91      0.93      0.92        46

    accuracy                           0.95       140
   macro avg       0.94      0.95      0.94       140
weighted avg       0.95      0.95      0.95       140
```

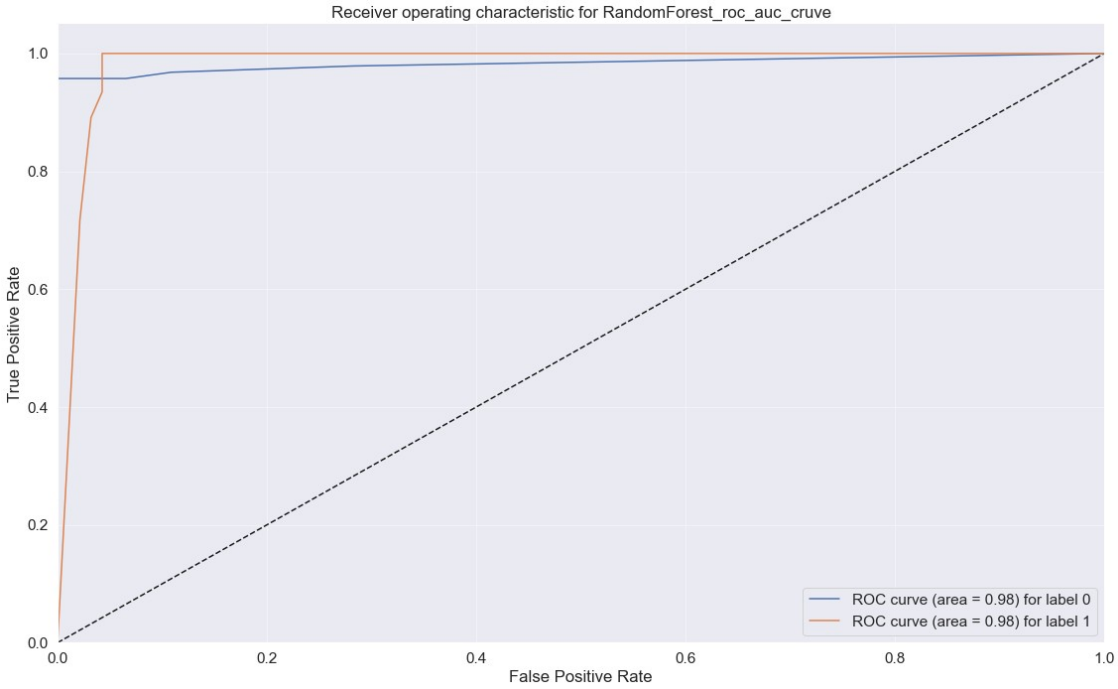
The precision and recall scores are higher for class 0, which is expected given that it's class that is twice the amount of class 1. However, the model is doing fairly well for predicting class 1. We can say that in terms of prediction for class 1, our model is correct 91% of the time. However, the the model was only able to predict 93% of all 46 True Positive instances (class 1).

```
roc_auc_score(val_target, knn_model.predict_proba(val_inputs)[:,1])
```

0.9833487511563368

```
plot_multiclass_roc(knn_model, val_inputs, val_target, 2,
figsize=(16,10))
```

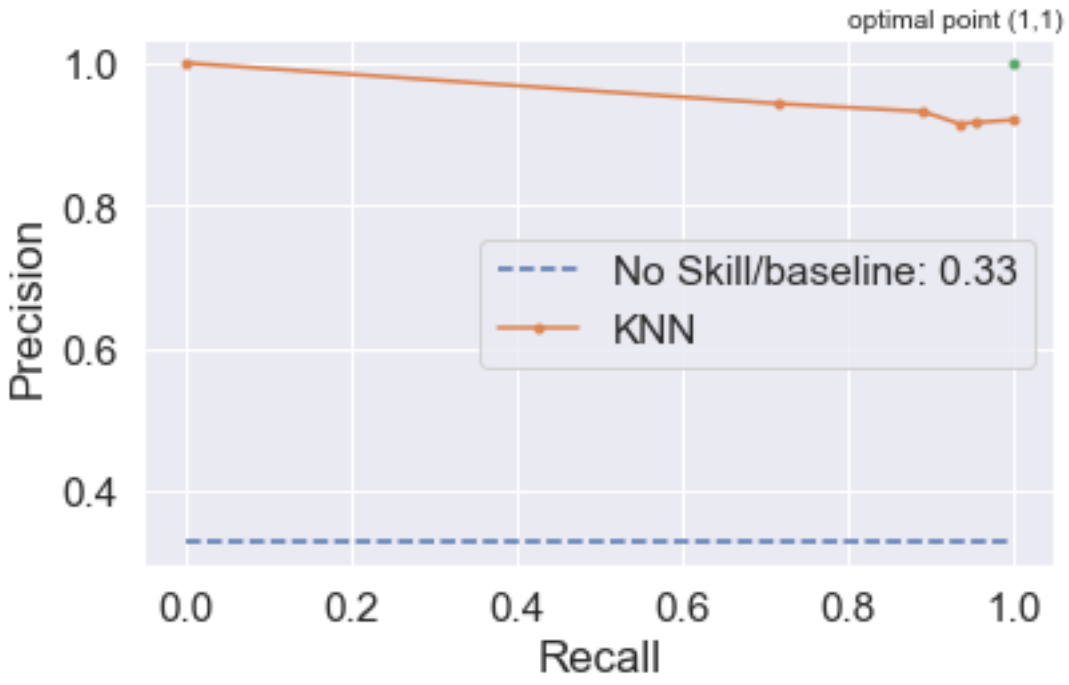
Using predict_proba method



ideal is 1.0, since we are at 0.98 for both classes, the model is performing well. Notice that the ROC curve and score for both of the classes are the similar (model isn't better at predicting No vs. Yes) despite having different recall and precision scores. It is worth noting that Recall for class 0 remains high all throughout the different thresholds, whereas, the recall for class 1 steadly drops from 1 to 0 as the line nears the maximum threshold value. I am somewhat suspicious of this graph, but based on our confusion matrix maybe the model is just performming really well. I would say the roc curve and aus score for it is a good indicator of the model's performance since the target values are not too heavily imabalanced.

```
pre_recall_curve(knn_model, val_inputs, val_target)
```

KNN: f1=0.925 auc=0.960



Usually, when precision decreases, recall increases... which is certainly the case here. Our F1 score and AUC score are both favorable. The different precision and recall scores are all looking very promising. I'd say this model is very good at predicting class 1. Persoanlly, I would choose a threshold that has greater recall than precision since False Negatives during breast cancer diagnosis is a better news than a False Positive. To reach a high recall score, it seems that we don't need to compromse that much precision, so that is very advantageous.

Hypertuning base KNN model

```
knn_random_grid = {
                'leaf_size':[n for n in range(1,20)],
                'n_neighbors':[n for n in range(1,51)],
                'p':[1,2],
                'weights':['uniform','distance']
                }

from sklearn.model_selection import GridSearchCV

knn_model_optimized = GridSearchCV(KNeighborsClassifier(n_jobs=-1),
knn_random_grid, cv=2, verbose=True, n_jobs=-1)

knn_model_optimized.fit(train_inputs, train_target)

Fitting 2 folds for each of 3800 candidates, totalling 7600 fits
```

```
GridSearchCV(cv=2, estimator=KNeighborsClassifier(n_jobs=-1), n_jobs=-
1,
             param_grid={'leaf_size': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12,
                                       13, 14, 15, 16, 17, 18, 19],
                         'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11, 12,
                                         13, 14, 15, 16, 17, 18, 19,
20, 21, 22,
                                         23, 24, 25, 26, 27, 28, 29,
30, ...],
                         'p': [1, 2], 'weights': ['uniform',
'distance']},
             verbose=True)
```

```
knn_model_optimized.best_params_
```

```
{'leaf_size': 1, 'n_neighbors': 3, 'p': 2, 'weights': 'uniform'}
```

```
knn_model_optimized.best_estimator_
```

```
KNeighborsClassifier(leaf_size=1, n_jobs=-1, n_neighbors=3)
```

```
knn_model.get_params()
```

```
{'algorithm': 'auto',
 'leaf_size': 30,
 'metric': 'minkowski',
 'metric_params': None,
 'n_jobs': -1,
 'n_neighbors': 5,
 'p': 2,
 'weights': 'uniform'}
```

It seems that there isn't too much of a difference between the parameters of our base and optimized model. leaf_size changed from 30 to 1, n_neighbors changed from 5 to 3.

```
val_pred_optimized = knn_model_optimized.predict(val_inputs)
```

Classification Metrics on Optimized KNN Model

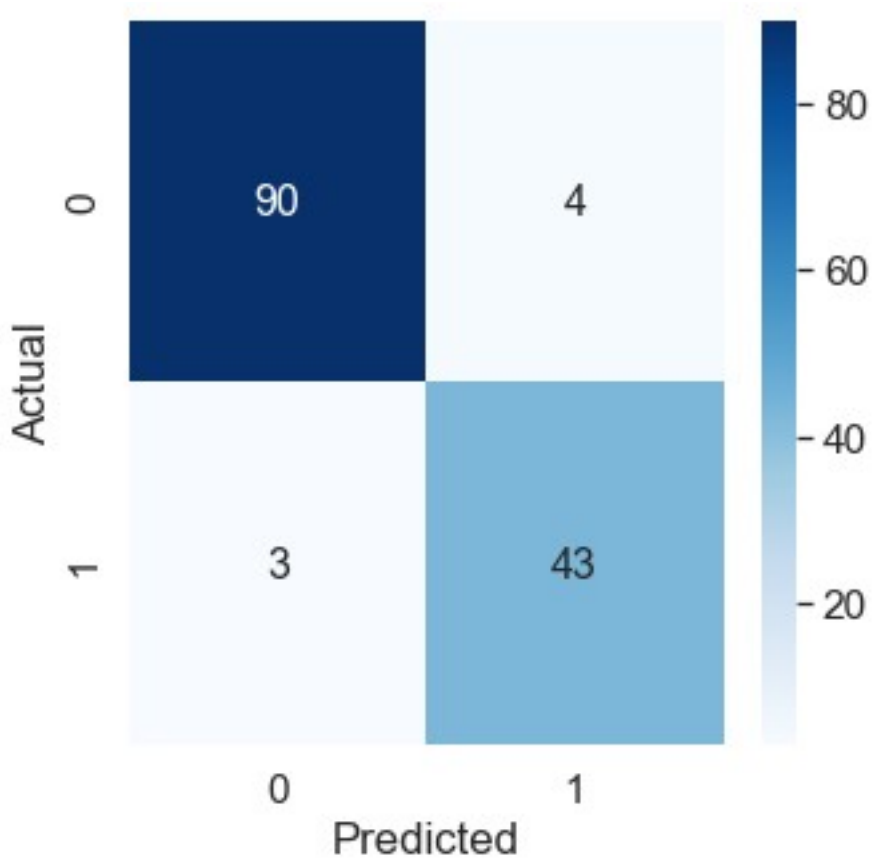
```
accuracyscores(knn_model_optimized, train_inputs, train_target,
val_inputs, val_target)
```

```
train model score 0.9809069212410502
non-train model score 0.95
random model accuracy score 0.5
majority model accuracy score 0.6714285714285714
```

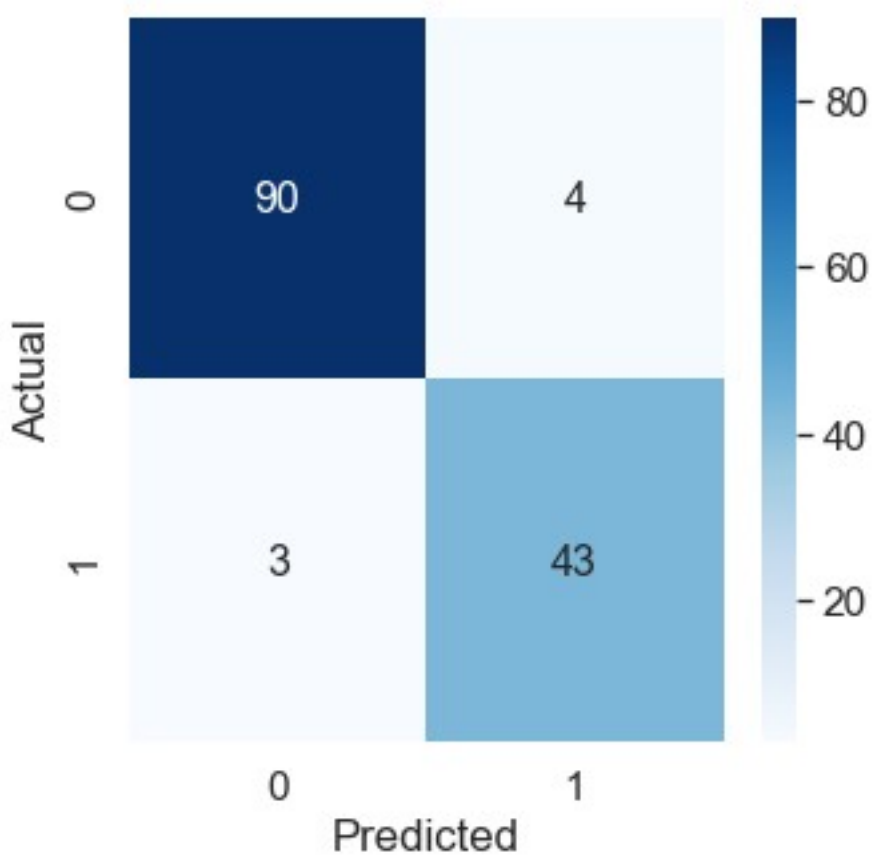
```
accuracyscores(knn_model, train_inputs, train_target, val_inputs,
val_target)
```

```
train model score 0.9737470167064439
non-train model score 0.95
random model accuracy score 0.4357142857142857
majority model accuracy score 0.6714285714285714
```

```
confusionmatrixplot(knn_model, val_inputs, val_target)
```




```
confusionmatrixplot(knn_model_optimized, val_inputs, val_target)
```




```
print(classification_report(val_target,
knn_model.predict(val_inputs)))
```

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96        94
           1       0.91      0.93      0.92        46

    accuracy                           0.95       140
   macro avg       0.94      0.95      0.94       140
weighted avg       0.95      0.95      0.95       140
```

```
print(classification_report(val_target,
knn_model_optimized.predict(val_inputs)))
```

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96        94
           1       0.91      0.93      0.92        46

    accuracy                           0.95       140
   macro avg       0.94      0.95      0.94       140
```

```
weighted avg       0.95      0.95      0.95       140
```

```
roc_auc_score(val_target, knn_model.predict_proba(val_inputs)[:,1])
```

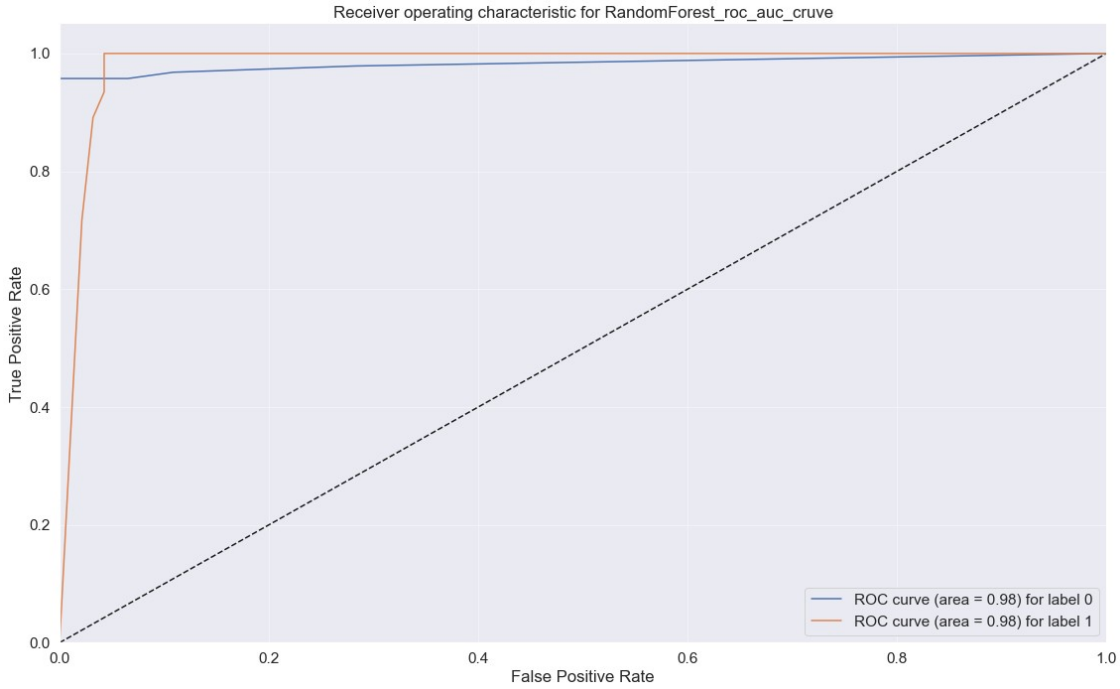
```
0.9833487511563368
```

```
roc_auc_score(val_target,
knn_model_optimized.predict_proba(val_inputs)[:,1])
```

```
0.9840425531914894
```

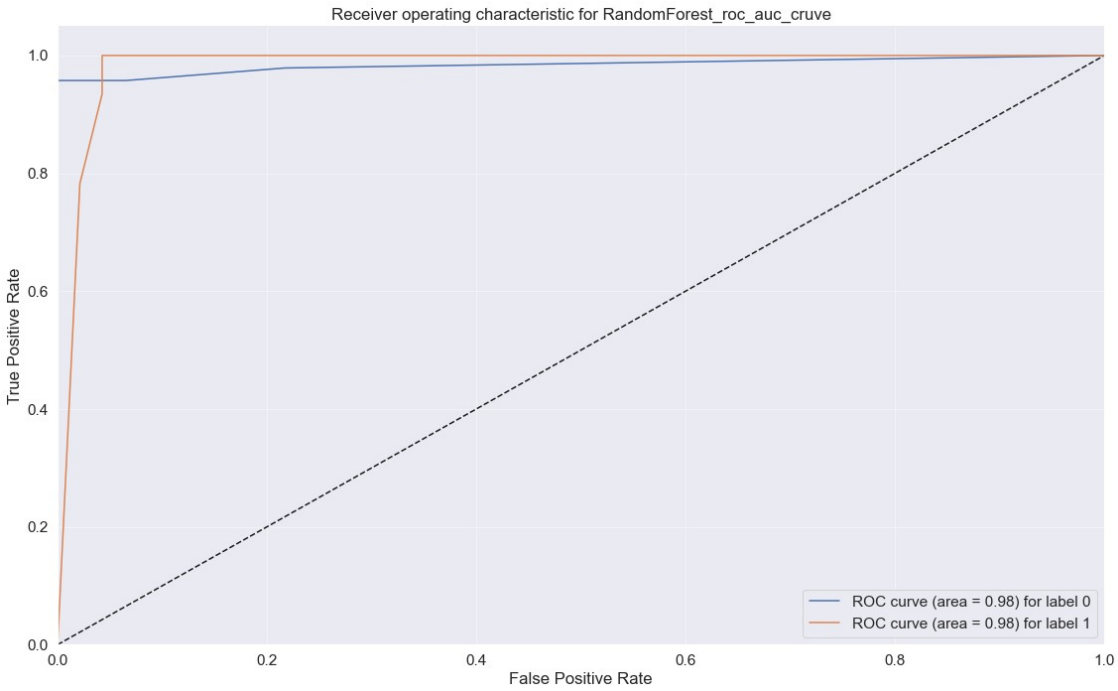
```
plot_multiclass_roc(knn_model, val_inputs, val_target, 2,
figsize=(16,10))
```

```
Using predict_proba method
```



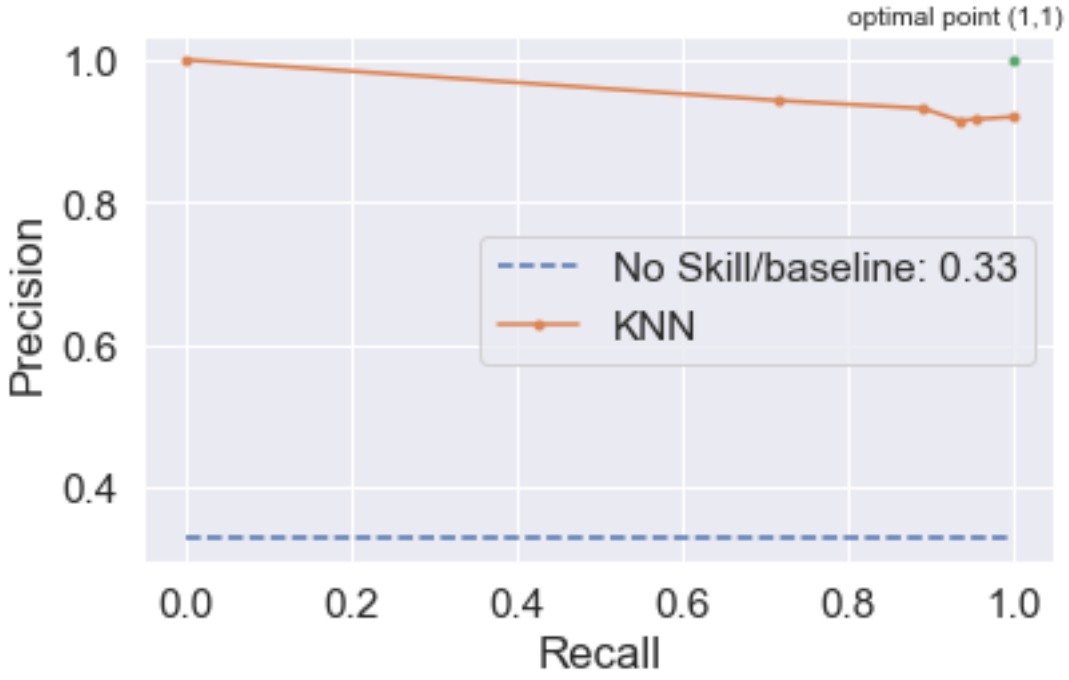

```
plot_multiclass_roc(knn_model_optimized, val_inputs, val_target, 2,
figsize=(16,10))
```

```
Using predict_proba method
```



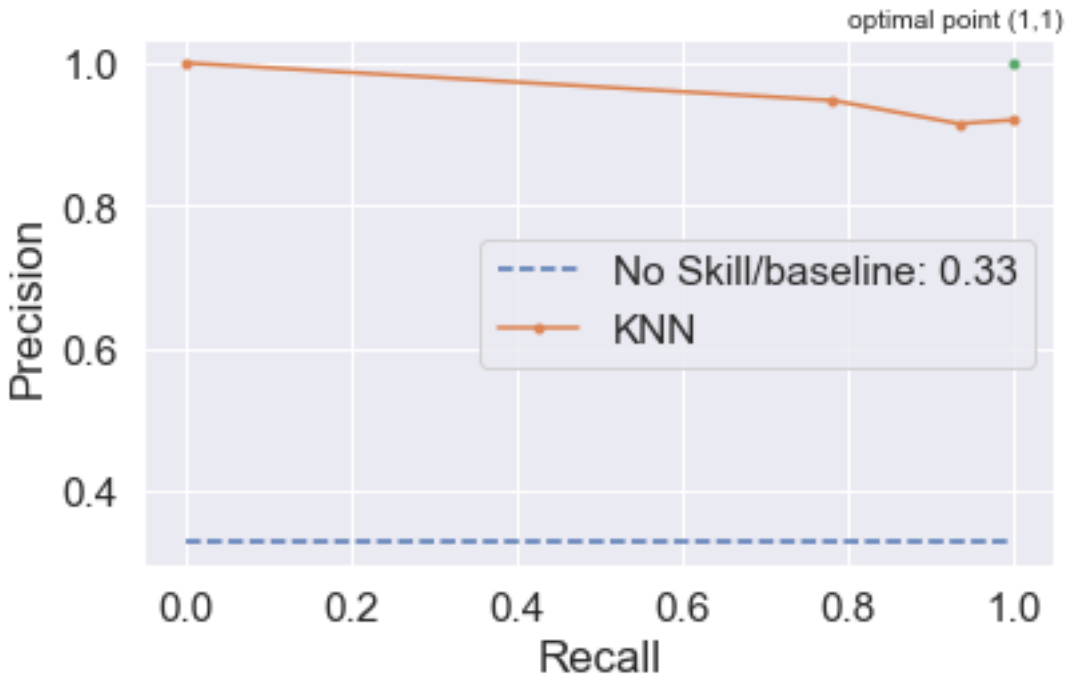

```
pre_recall_curve(knn_model, val_inputs, val_target)
```

```
KNN: f1=0.925 auc=0.960
```




```
pre_recall_curve(knn_model_optimized, val_inputs, val_target)
```

```
KNN: f1=0.925 auc=0.964
```



`Classification Report`

- Optimized Model hasn't improved the precision, recall, and F1 scores for both classes compared to the base model.

`Confusion Matrix`

- Number of False Negatives and False Positves are the same for the base and optimized KNN model.

`Accuracy Score`

- Accuracy score for the optimized modeal was higher than the base model when working with the train dataset, but are otherwise the same when it came to the validation set.

`ROC AUC Score`

- AUC score for the optimized model is higher by a slight, negligible margin.

`Precision vs Recall curve and AUC score`

- AUC score for the optimized model is higher by a slight, negligible margin when it came to the precision vs recall curve.

Both Models perform the same, but since the purpose of the model is to predict breast cancer, I'd go for the optimized model.

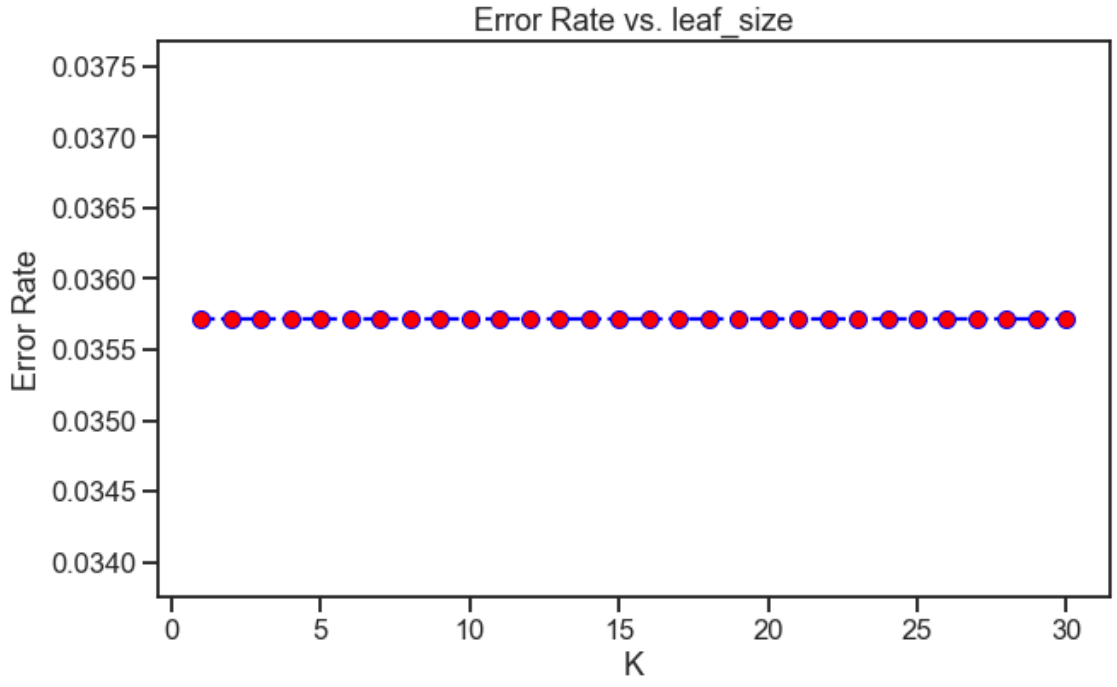
Error rate and accuracy rate w/ different k-values

```
error_rate = []


for i in range(1,31):

    knn = KNeighborsClassifier(leaf_size=i, n_jobs = -1,
n_neighbors=7)
    knn.fit(train_inputs,train_target)
    pred_i = knn.predict(val_inputs)
    error_rate.append(np.mean(pred_i != val_target))
```

Create a visualization to compare the error rate and k value

```
plt.figure(figsize=(10,6))
plt.plot(range(1,31),error_rate,color='blue', linestyle='dashed',
marker='o',
         markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. leaf_size')
plt.xlabel('K')
plt.ylabel('Error Rate')
```

```
Text(0, 0.5, 'Error Rate')
```



Number of leaf vs error rate

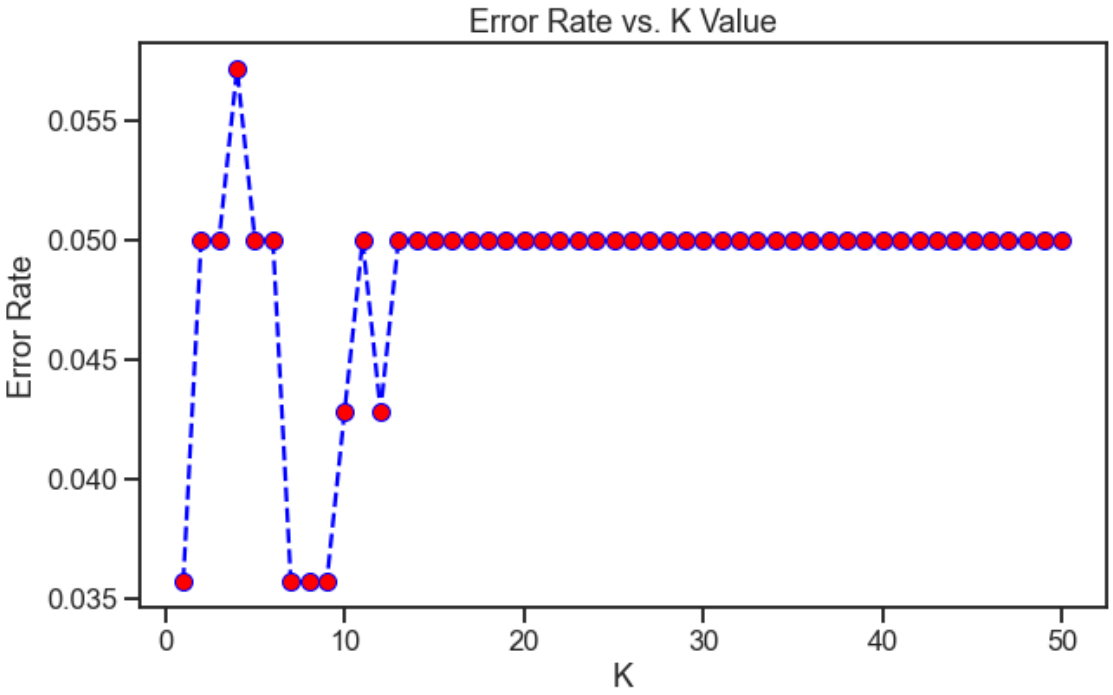
```
error_rate = []
```

```
for i in range(1,51):

    knn = KNeighborsClassifier(n_neighbors=i, n_jobs = -1)
    knn.fit(train_inputs,train_target)
    pred_i = knn.predict(val_inputs)
    error_rate.append(np.mean(pred_i != val_target))
```

Create a visualization to compare the error rate and k value

```
plt.figure(figsize=(10,6))
plt.plot(range(1,51),error_rate,color='blue', linestyle='dashed',
marker='o',
         markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
```

```
Text(0, 0.5, 'Error Rate')
```




```
knn = KNeighborsClassifier(leaf_size=1, n_jobs = -1, n_neighbors=7)
knn.fit(train_inputs,train_target)
knn_predict = knn.predict(val_inputs)
accuracyscores(knn, train_inputs, train_target, val_inputs,
val_target)
```

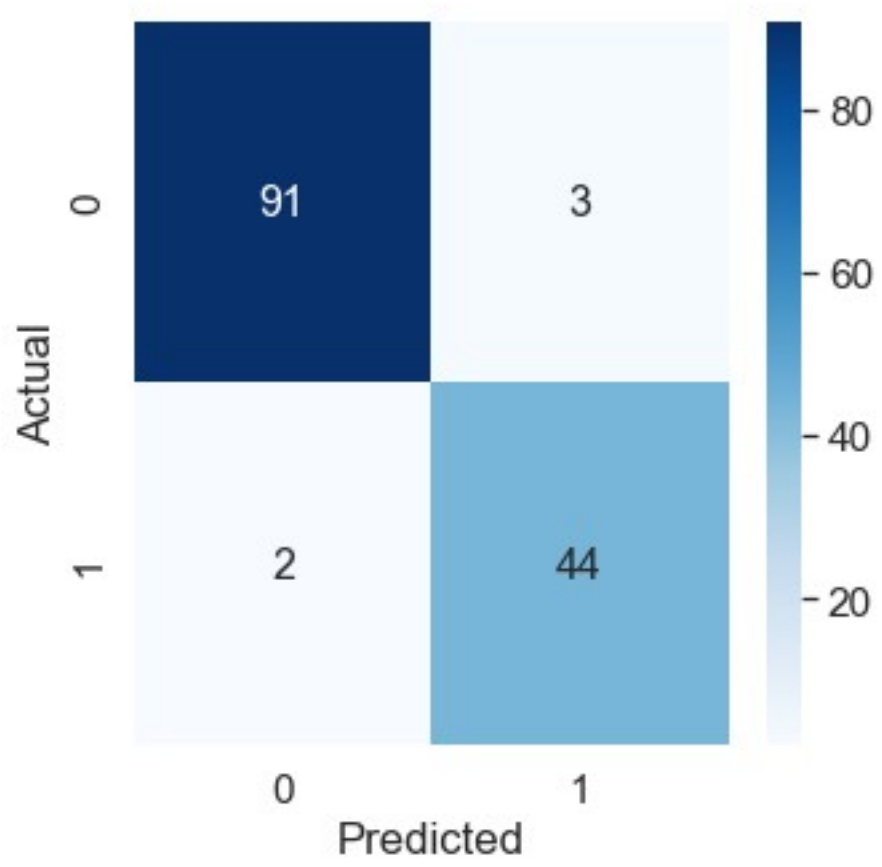
```
train model score 0.9761336515513126
non-train model score 0.9642857142857143
random model accuracy score 0.45
majority model accuracy score 0.6714285714285714
```

```
print(classification_report(val_target, knn_predict))
```

```
              precision    recall  f1-score   support

           0       0.98      0.97      0.97        94
           1       0.94      0.96      0.95        46

    accuracy                           0.96       140
   macro avg       0.96      0.96      0.96       140
weighted avg       0.96      0.96      0.96       140
```

```
print(confusionmatrixplot(knn, val_inputs, val_target))
```

```
None
```

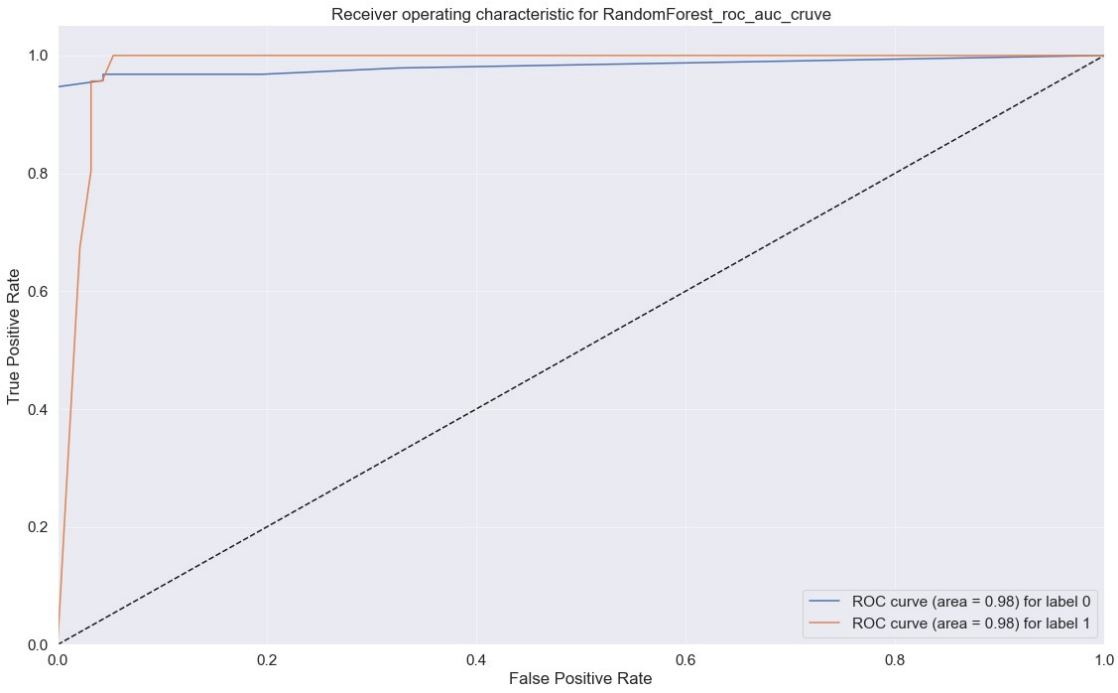



```
print(roc_auc_score(val_target, knn.predict_proba(val_inputs)[:,1]))
```

```
0.9824236817761332
```

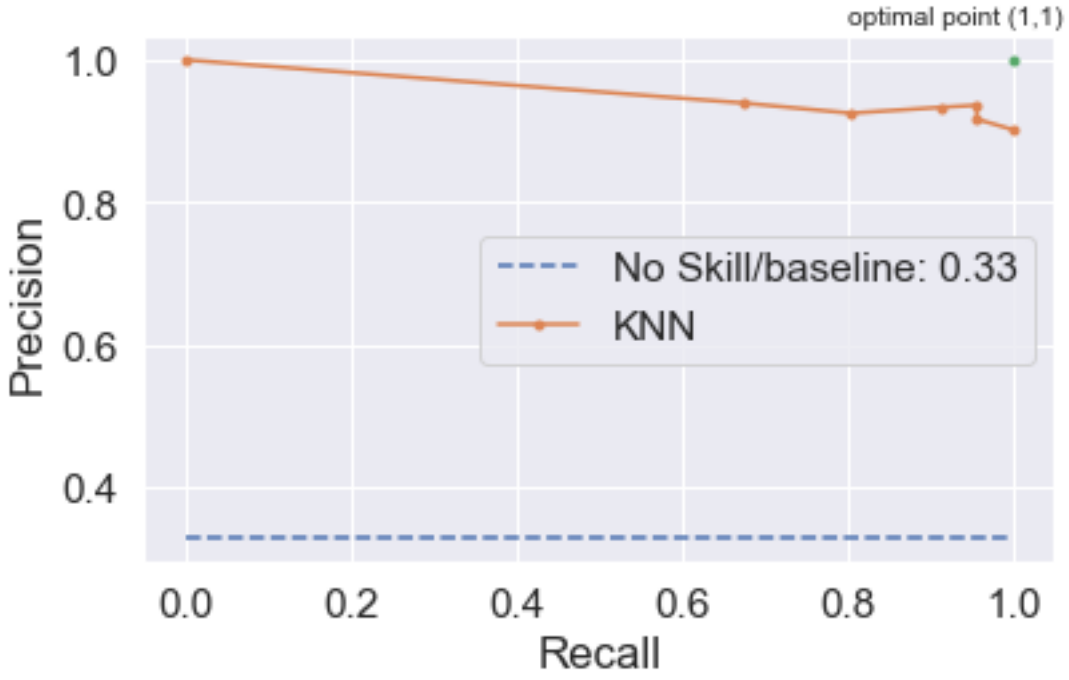
```
plot_multiclass_roc(knn, val_inputs, val_target, 2, figsize=(16,10))
```

```
Using predict_proba method
```




```
pre_recall_curve(knn, val_inputs, val_target)
```

```
KNN: f1=0.946 auc=0.956
```




```
Classification Report
```

- Optimized

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96        94
           1       0.91      0.93      0.92        46
```

- n_neighbors = 7

```
              precision    recall  f1-score   support

           0       0.98      0.97      0.97        94
           1       0.94      0.96      0.95        46
```

`Confusion Matrix`

- Optimized 4 False Positive and 3 False Negatives
- model w/ n_neighbors = 7 3 False Positive and 2 False Negatives

`Accuracy Score`

- Optimized: 95%
- model w/ n_neighbors 7: 96.42857142857143%

`ROC AUC Score`

- Optimized: 0.9840425531914894
- model w/ n_neighbors =7: 0.9824236817761332

`Precision vs Recall curve and AUC score`

- optimized - KNN: f1=0.925 auc=0.964
- model w/ n_neighbors = 7 - KNN: f1=0.946 auc=0.956

`Model with n_neighbors = 7 performs better than the optimized model in the metrics of `classification report`, `confusion matrix`, and `accuracy score`.`

`Marginally worse in the AUC scores for both `ROC` and `precision vs recall` curve`

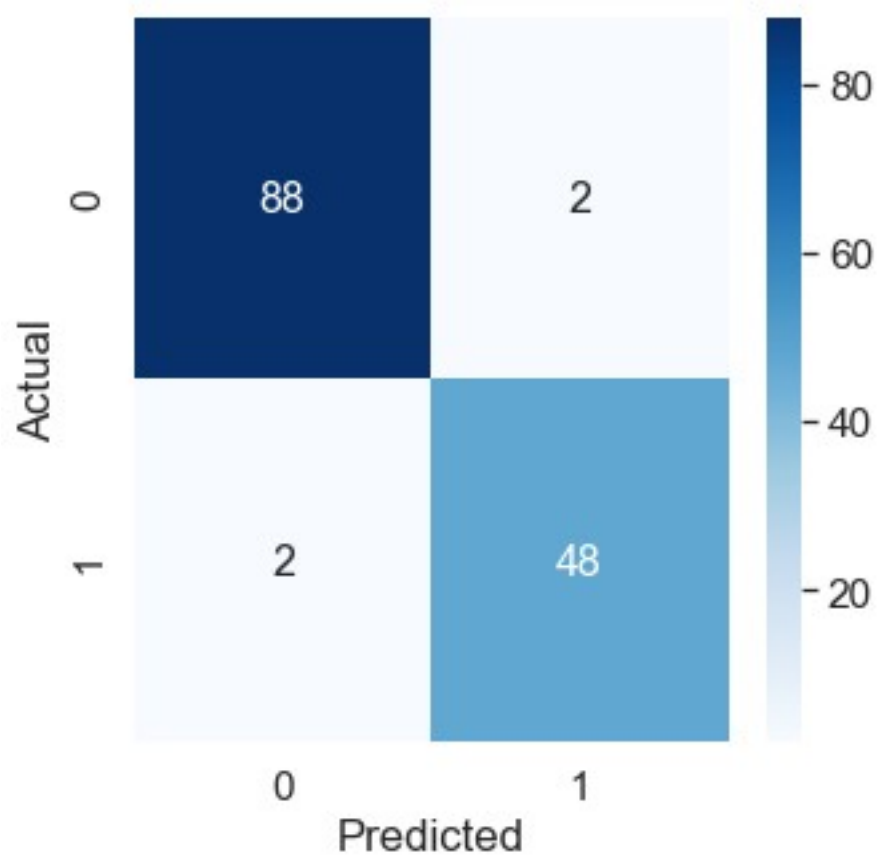
Test Dataset Metrics

Use model that was optimzed using grid searc VS the model that has n_neighbors = 7

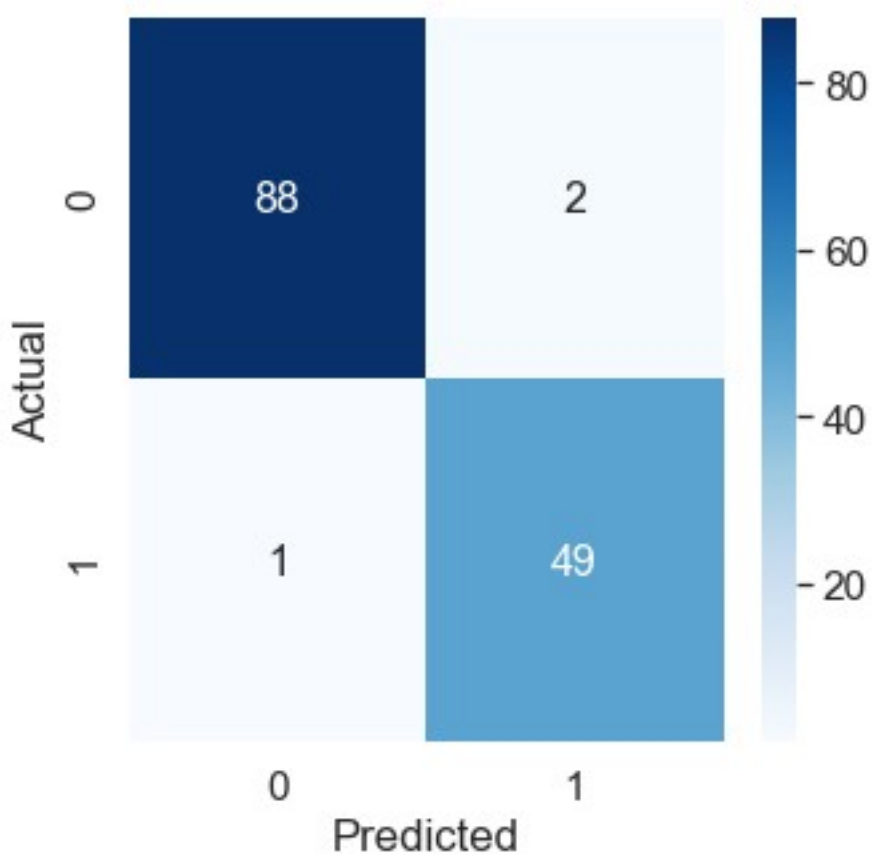
```
accuracyscores(knn_model_optimized, train_inputs, train_target, test_inputs, test_target)
```

```
train model score 0.9809069212410502
non-train model score 0.9714285714285714
random model accuracy score 0.5428571428571428
majority model accuracy score 0.6428571428571429
```

```
accuracyscores(knn, train_inputs, train_target, test_inputs,
test_target)

train model score 0.9761336515513126
non-train model score 0.9785714285714285
random model accuracy score 0.45714285714285713
majority model accuracy score 0.6428571428571429

confusionmatrixplot(knn_model_optimized, test_inputs, test_target)
```




```
confusionmatrixplot(knn, test_inputs, test_target)
```




```
print(classification_report(test_target,
knn_model_optimized.predict(test_inputs)))
```

```
              precision    recall  f1-score   support

           0       0.98      0.98      0.98        90
           1       0.96      0.96      0.96        50

    accuracy                           0.97       140
   macro avg       0.97      0.97      0.97       140
weighted avg       0.97      0.97      0.97       140
```

```
print(classification_report(test_target, knn.predict(test_inputs)))
```

```
              precision    recall  f1-score   support

           0       0.99      0.98      0.98        90
           1       0.96      0.98      0.97        50

    accuracy                           0.98       140
   macro avg       0.97      0.98      0.98       140
weighted avg       0.98      0.98      0.98       140
```

```
roc_auc_score(test_target,
knn_model_optimized.predict_proba(test_inputs)[:,1])
```

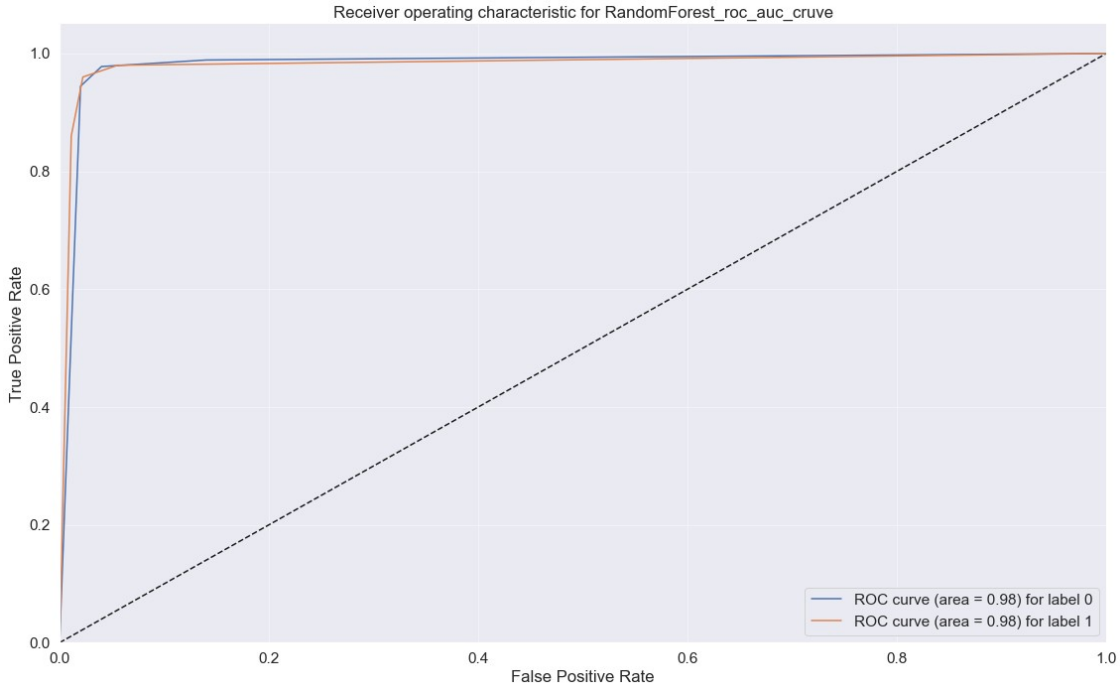
```
0.9822222222222221
```

```
roc_auc_score(test_target, knn.predict_proba(test_inputs)[:,1])
```

```
0.9925555555555556
```

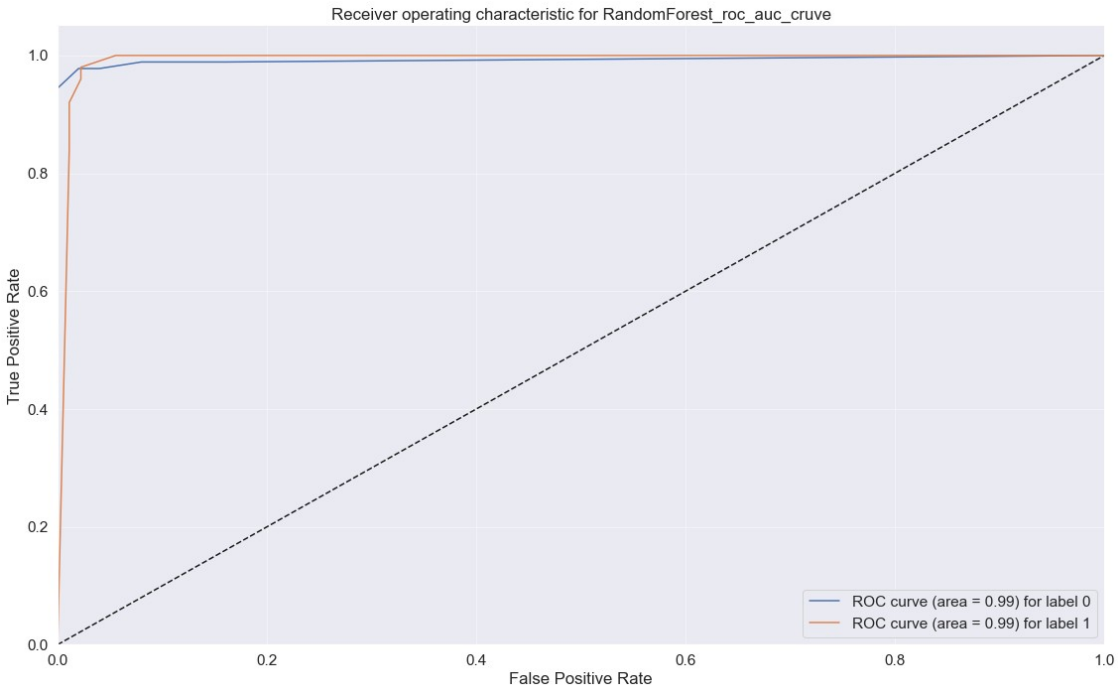
```
plot_multiclass_roc(knn_model_optimized, test_inputs, test_target, 2,
figsize=(16,10))
```

```
Using predict_proba method
```



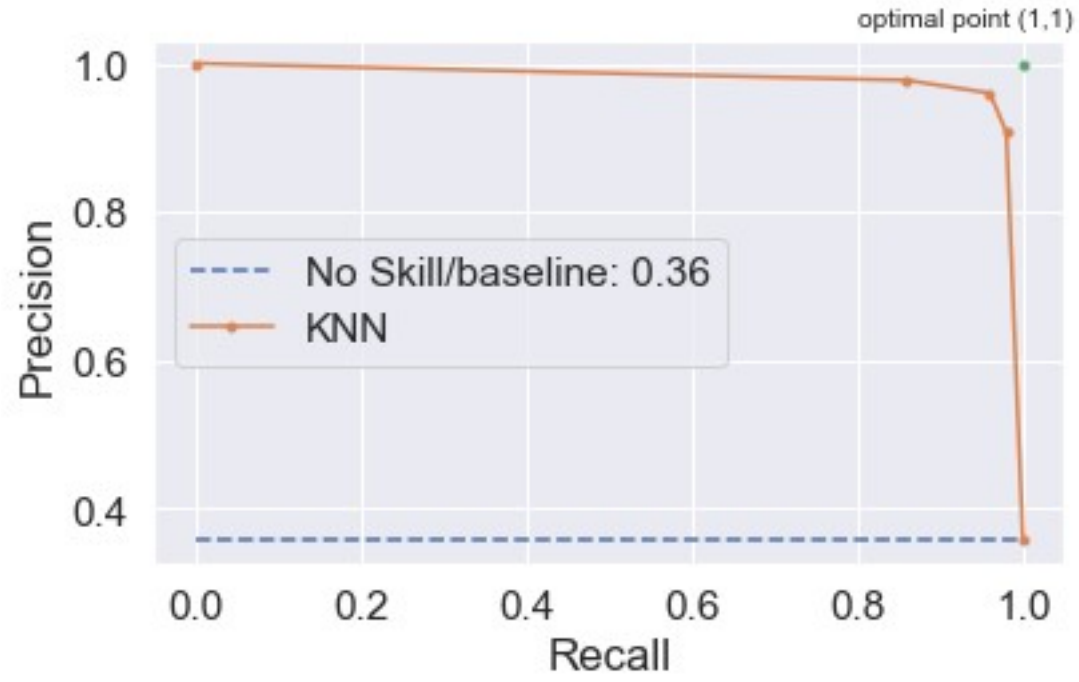

```
plot_multiclass_roc(knn, test_inputs, test_target, 2, figsize=(16,10))
```

```
Using predict_proba method
```



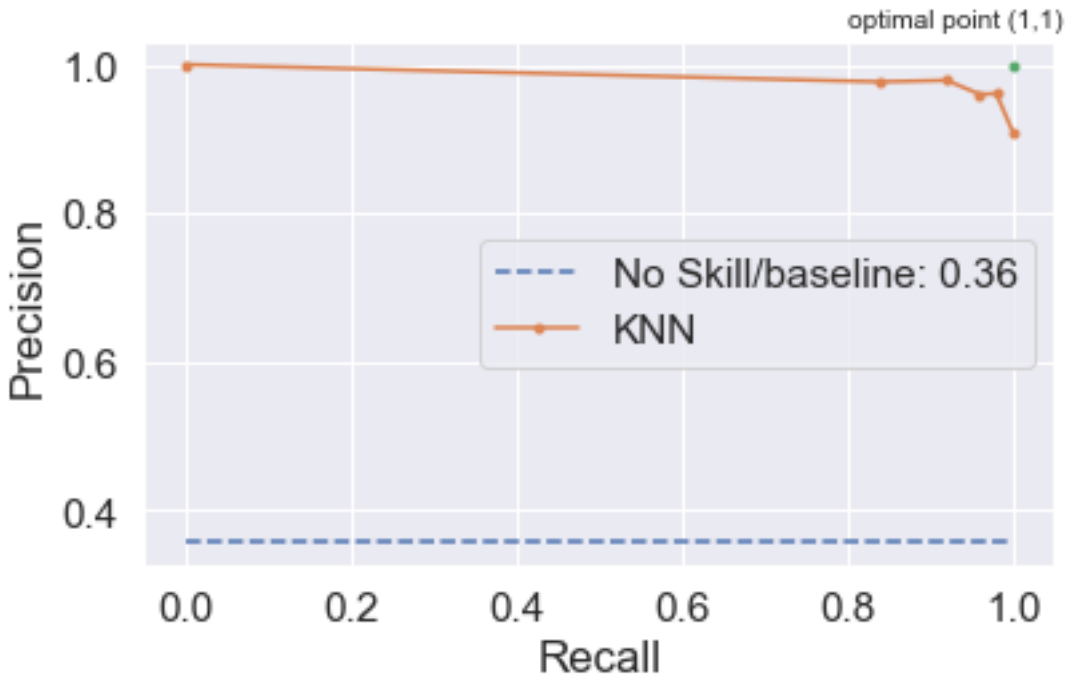

```
pre_recall_curve(knn_model_optimized, test_inputs, test_target)
```

```
KNN: f1=0.960 auc=0.978
```




```
pre_recall_curve(knn, test_inputs, test_target)
```

```
KNN: f1=0.970 auc=0.985
```



see observation summary in conclusion section

Conclusion

Our model with the k_neighbors = 7 performs better than the optimized and base model in the metrics of `classification report`, `confusion matrix`, `accuracy score`,`ROC AUC score`, and `precision vs recall AUC score` curve.

Without a doubt the model called `knn` with the specific paramter of `n_neighbors = 7` is the model that is best and is most approproate to use in the real-world/prduction