theAwesome_PredModel

May 3, 2017

1 Prediction Model

- **Course**: Data Mining
- **Team/StudentName**: The Awesome
- **Team Members**: Enes, Kemal, Ergin
- **Member Contribution:**
  - **Enes**: Steps 1-4
  - **Kemal**: Steps 5-7
  - **Ergin**: Steps 8-10

1.1 Step 0: Data Preparation and Cleaning

```
In [1]: import pandas as pd
```

```
In [2]: # Read CSV data into df
        df = pd.read_csv('./theAwesome_PredModel.csv')
        # delete id column no need
        df.drop('id',axis=1,inplace=True)
        # delete unnamed colum at the end
        df.drop('Unnamed: 32',axis=1,inplace=True)
        df.head()
```

```
Out[2]:   diagnosis  radius_mean  texture_mean  perimeter_mean  area_mean  \
        0         M        17.99         10.38          122.80     1001.0
        1         M        20.57         17.77          132.90     1326.0
        2         M        19.69         21.25          130.00     1203.0
        3         M        11.42         20.38           77.58      386.1
        4         M        20.29         14.34          135.10     1297.0

           smoothness_mean  compactness_mean  concavity_mean  concave points_mean  \
        0          0.11840           0.27760          0.3001              0.14710
        1          0.08474           0.07864          0.0869              0.07017
        2          0.10960           0.15990          0.1974              0.12790
        3          0.14250           0.28390          0.2414              0.10520
        4          0.10030           0.13280          0.1980              0.10430

           symmetry_mean         ...           radius_worst  texture_worst  \
```

```
0        0.2419    ...                25.38          17.33
1        0.1812    ...                24.99          23.41
2        0.2069    ...                23.57          25.53
3        0.2597    ...                14.91          26.50
4        0.1809    ...                22.54          16.67

   perimeter_worst  area_worst  smoothness_worst  compactness_worst  \
0           184.60      2019.0            0.1622             0.6656
1           158.80      1956.0            0.1238             0.1866
2           152.50      1709.0            0.1444             0.4245
3            98.87       567.7            0.2098             0.8663
4           152.20      1575.0            0.1374             0.2050

   concavity_worst  concave points_worst  symmetry_worst  \
0           0.7119                0.2654          0.4601
1           0.2416                0.1860          0.2750
2           0.4504                0.2430          0.3613
3           0.6869                0.2575          0.6638
4           0.4000                0.1625          0.2364

   fractal_dimension_worst
0                  0.11890
1                  0.08902
2                  0.08758
3                  0.17300
4                  0.07678

[5 rows x 31 columns]
```

In [3]:
```python
# Learn the unique values in diagnosis column
df.diagnosis.unique()
# M: Malign (Yes Cancer)
# B: Benign (No Cancer)

# I can also map M and B as 1 and 0 for more numerical
#  approach
df['diagnosis'] = df['diagnosis'].map({'M':1,'B':0})
```

1.2 Step 1: Data Information and Descriptive Statistics

Generate the information about your dataset: number of columns and rows, names and data types of the columns, memory usage of the dataset.

Hint: Pandas data frame info() function.

Generate descriptive statistics of all columns (input and output) of your dataset. Descriptive statistics for numerical columns include: count, mean, std, min, 25 percentile (Q1), 50 percentile (Q2, median), 75 percentile (Q3), max values of the columns. For categorical columns, determine distinct values and their frequency in each categorical column.

Hint: Pandas, data frame describe() function.

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 31 columns):
diagnosis                  569 non-null int64
radius_mean                569 non-null float64
texture_mean               569 non-null float64
perimeter_mean             569 non-null float64
area_mean                  569 non-null float64
smoothness_mean            569 non-null float64
compactness_mean           569 non-null float64
concavity_mean             569 non-null float64
concave points_mean        569 non-null float64
symmetry_mean              569 non-null float64
fractal_dimension_mean     569 non-null float64
radius_se                  569 non-null float64
texture_se                 569 non-null float64
perimeter_se               569 non-null float64
area_se                    569 non-null float64
smoothness_se              569 non-null float64
compactness_se             569 non-null float64
concavity_se               569 non-null float64
concave points_se          569 non-null float64
symmetry_se                569 non-null float64
fractal_dimension_se       569 non-null float64
radius_worst               569 non-null float64
texture_worst              569 non-null float64
perimeter_worst            569 non-null float64
area_worst                 569 non-null float64
smoothness_worst           569 non-null float64
compactness_worst          569 non-null float64
concavity_worst            569 non-null float64
concave points_worst       569 non-null float64
symmetry_worst             569 non-null float64
fractal_dimension_worst    569 non-null float64
dtypes: float64(30), int64(1)
memory usage: 137.9 KB
```

```
In [5]: df.describe()
```

```
Out[5]:         diagnosis  radius_mean  texture_mean  perimeter_mean   area_mean  \
        count  569.000000   569.000000    569.000000      569.000000  569.000000
        mean     0.372583    14.127292     19.289649       91.969033  654.889104
        std      0.483918     3.524049      4.301036       24.298981  351.914129
        min      0.000000     6.981000      9.710000       43.790000  143.500000
```

25%	0.000000	11.700000	16.170000	75.170000	420.300000
50%	0.000000	13.370000	18.840000	86.240000	551.100000
75%	1.000000	15.780000	21.800000	104.100000	782.700000
max	1.000000	28.110000	39.280000	188.500000	2501.000000

	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	\
count	569.000000	569.000000	569.000000	569.000000	
mean	0.096360	0.104341	0.088799	0.048919	
std	0.014064	0.052813	0.079720	0.038803	
min	0.052630	0.019380	0.000000	0.000000	
25%	0.086370	0.064920	0.029560	0.020310	
50%	0.095870	0.092630	0.061540	0.033500	
75%	0.105300	0.130400	0.130700	0.074000	
max	0.163400	0.345400	0.426800	0.201200	

	symmetry_mean	...	radius_worst	texture_worst	\
count	569.000000	...	569.000000	569.000000	
mean	0.181162	...	16.269190	25.677223	
std	0.027414	...	4.833242	6.146258	
min	0.106000	...	7.930000	12.020000	
25%	0.161900	...	13.010000	21.080000	
50%	0.179200	...	14.970000	25.410000	
75%	0.195700	...	18.790000	29.720000	
max	0.304000	...	36.040000	49.540000	

	perimeter_worst	area_worst	smoothness_worst	compactness_worst	\
count	569.000000	569.000000	569.000000	569.000000	
mean	107.261213	880.583128	0.132369	0.254265	
std	33.602542	569.356993	0.022832	0.157336	
min	50.410000	185.200000	0.071170	0.027290	
25%	84.110000	515.300000	0.116600	0.147200	
50%	97.660000	686.500000	0.131300	0.211900	
75%	125.400000	1084.000000	0.146000	0.339100	
max	251.200000	4254.000000	0.222600	1.058000	

	concavity_worst	concave points_worst	symmetry_worst	\
count	569.000000	569.000000	569.000000	
mean	0.272188	0.114606	0.290076	
std	0.208624	0.065732	0.061867	
min	0.000000	0.000000	0.156500	
25%	0.114500	0.064930	0.250400	
50%	0.226700	0.099930	0.282200	
75%	0.382900	0.161400	0.317900	
max	1.252000	0.291000	0.663800	

	fractal_dimension_worst
count	569.000000
mean	0.083946

```
std                    0.018061
min                    0.055040
25%                    0.071460
50%                    0.080040
75%                    0.092080
max                    0.207500

[8 rows x 31 columns]
```

1.3 Step 2: Train Test Split

Split your data into Training and Test data set by randomly selecting; use 70% for training and 30 % for testing. Generate descriptive statistics of all columns (input and output) of Training and Test datasets. Review the descriptive statistics of input output columns in Train, Test and original Full (before the splitting operation) datasets and compare them to each other. Are they similar or not? Do you think Train and Test dataset are representative of the Full datasets ? why ?

Hint: Scikit learn, data train_test_split(), stratified function.

In [6]:
```
df["diagnosis"].value_counts(df["diagnosis"].unique()[0])
```

Out[6]:
```
0    0.627417
1    0.372583
Name: diagnosis, dtype: float64
```

In [7]:
```
# Splitting train and test data
# .7 and .3
import numpy as np # Linear algebra and numerical apps
msk = np.random.rand(len(df)) < 0.7
train_df = df[msk]
test_df = df[~msk]
```

1.4 Step 3: Analysis of the Output Column

Analyze the output columns in Train and Test dataset. If the output column is numerical then calculate the IQR (inter quartile range, Q3-Q1) and Range (difference between max and min value). If your output column is categorical then determine if the column is nominal or ordinal, why?. Is there a class imbalance problem? (check if there is big difference between the number of distinct values in your categorical output column)

In [8]:
```
print(train_df["diagnosis"].value_counts(train_df["diagnosis"].unique()[0]))
print(len(train_df))
train_df.head()
```

```
0    0.631325
1    0.368675
Name: diagnosis, dtype: float64
415
```

```
Out[8]:    diagnosis  radius_mean  texture_mean  perimeter_mean  area_mean  \
        1          1        20.57         17.77          132.90     1326.0
        2          1        19.69         21.25          130.00     1203.0
        3          1        11.42         20.38           77.58      386.1
        5          1        12.45         15.70           82.57      477.1
        8          1        13.00         21.82           87.50      519.8

           smoothness_mean  compactness_mean  concavity_mean  concave points_mean  \
        1          0.08474           0.07864          0.0869              0.07017
        2          0.10960           0.15990          0.1974              0.12790
        3          0.14250           0.28390          0.2414              0.10520
        5          0.12780           0.17000          0.1578              0.08089
        8          0.12730           0.19320          0.1859              0.09353

           symmetry_mean          ...            radius_worst  texture_worst  \
        1         0.1812          ...                   24.99          23.41
        2         0.2069          ...                   23.57          25.53
        3         0.2597          ...                   14.91          26.50
        5         0.2087          ...                   15.47          23.75
        8         0.2350          ...                   15.49          30.73

           perimeter_worst  area_worst  smoothness_worst  compactness_worst  \
        1           158.80      1956.0            0.1238             0.1866
        2           152.50      1709.0            0.1444             0.4245
        3            98.87       567.7            0.2098             0.8663
        5           103.40       741.6            0.1791             0.5249
        8           106.20       739.3            0.1703             0.5401

           concavity_worst  concave points_worst  symmetry_worst  \
        1           0.2416                0.1860          0.2750
        2           0.4504                0.2430          0.3613
        3           0.6869                0.2575          0.6638
        5           0.5355                0.1741          0.3985
        8           0.5390                0.2060          0.4378

           fractal_dimension_worst
        1                  0.08902
        2                  0.08758
        3                  0.17300
        5                  0.12440
        8                  0.10720

        [5 rows x 31 columns]
```

```
In [9]: print(test_df["diagnosis"].value_counts(test_df["diagnosis"].unique()[0]))
        print(len(test_df))
        test_df.head()
```

```
0    0.616883
```

```
1    0.383117
Name: diagnosis, dtype: float64
154
```

Out[9]:

```
    diagnosis  radius_mean  texture_mean  perimeter_mean  area_mean  \
0           1        17.99         10.38           122.8     1001.0
4           1        20.29         14.34           135.1     1297.0
6           1        18.25         19.98           119.6     1040.0
7           1        13.71         20.83            90.2      577.9
17          1        16.13         20.68           108.1      798.8

    smoothness_mean  compactness_mean  concavity_mean  concave points_mean  \
0           0.11840            0.2776         0.30010              0.14710
4           0.10030            0.1328         0.19800              0.10430
6           0.09463            0.1090         0.11270              0.07400
7           0.11890            0.1645         0.09366              0.05985
17          0.11700            0.2022         0.17220              0.10280

    symmetry_mean          ...             radius_worst  texture_worst  \
0          0.2419          ...                    25.38          17.33
4          0.1809          ...                    22.54          16.67
6          0.1794          ...                    22.88          27.66
7          0.2196          ...                    17.06          28.14
17         0.2164          ...                    20.96          31.48

    perimeter_worst  area_worst  smoothness_worst  compactness_worst  \
0             184.6      2019.0            0.1622             0.6656
4             152.2      1575.0            0.1374             0.2050
6             153.2      1606.0            0.1442             0.2576
7             110.6       897.0            0.1654             0.3682
17            136.8      1315.0            0.1789             0.4233

    concavity_worst  concave points_worst  symmetry_worst  \
0            0.7119                0.2654          0.4601
4            0.4000                0.1625          0.2364
6            0.3784                0.1932          0.3063
7            0.2678                0.1556          0.3196
17           0.4784                0.2073          0.3706

    fractional_dimension_worst
0                      0.11890
4                      0.07678
6                      0.08368
7                      0.11510
17                     0.11420

[5 rows x 31 columns]
```

Our output/classification label is diagnosis(M(1)/B(0)), which is nominal categorical data.

The ratios between Benign and Malignant outputs in train and test are pretty similar to what we had in the full data.

1.5 Step 4: Scale Training and Test Dataset

Using one of the scaling method (max, min-max, standard or robust), create a scaler object and scale the numerical input columns of the Training dataset. Using the same scaler object, scale the numerical input columns of the Test set. Generate the descriptive statistics of the scaled input columns of Training and Test set.

If some of the input columns are categorical then convert them to binary columns using onehotencoder() function (scikit learn) or dummy() function (Pandas data frame).

Hint: http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing

```
In [10]: # I am going to apply min-max scaling for my data.
         from sklearn import preprocessing
         # Fitting the minmax scaled version for training data
         minmax_scale = preprocessing.MinMaxScaler().fit(train_df.iloc[:, 1:])
         # Now actually scale train and test data
         train_df.iloc[:, 1:] = minmax_scale.transform(train_df.iloc[:, 1:])
         test_df.iloc[:, 1:] = minmax_scale.transform(test_df.iloc[:, 1:])
```

```
/Users/eneskemalergin/anaconda3/lib/python3.5/site-packages/pandas/core/indexing.py:477: Setting
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#
  self.obj[item] = s
```

```
In [11]: train_df.head()
```

```
Out[11]:    diagnosis  radius_mean  texture_mean  perimeter_mean  area_mean  \
         1          1     0.643144      0.272574        0.615783   0.501591
         2          1     0.601496      0.390260        0.595743   0.449417
         3          1     0.210090      0.360839        0.233501   0.102906
         5          1     0.258839      0.202570        0.267984   0.141506
         8          1     0.284869      0.409537        0.302052   0.159618

            smoothness_mean  compactness_mean  concavity_mean  concave points_mean  \
         1         0.277910          0.181768        0.203799             0.366806
         2         0.588699          0.431017        0.462946             0.668583
         3         1.000000          0.811361        0.566135             0.549922
         5         0.816227          0.461996        0.370075             0.422844
         8         0.809976          0.533157        0.435976             0.488918

            symmetry_mean           ...             radius_worst  texture_worst  \
```

```
1       0.407367            ...                   0.606901       0.324132
2       0.546587            ...                   0.556386       0.384462
3       0.832611            ...                   0.248310       0.412066
5       0.556338            ...                   0.268232       0.333808
8       0.698808            ...                   0.268943       0.532442

   perimeter_worst  area_worst  smoothness_worst  compactness_worst  \
1         0.539818    0.435214          0.301026           0.148757
2         0.508442    0.374508          0.446763           0.381154
3         0.241347    0.094008          0.909445           0.812734
5         0.263908    0.136748          0.692253           0.479232
8         0.277852    0.136183          0.629996           0.494080

   concavity_worst  concave points_worst  symmetry_worst  \
1         0.192971              0.639175        0.233590
2         0.359744              0.835052        0.403706
3         0.548642              0.884880        1.000000
5         0.427716              0.598282        0.477035
8         0.430511              0.707904        0.554504

   fractal_dimension_worst
1                 0.222878
2                 0.213433
3                 0.773711
5                 0.454939
8                 0.342123

[5 rows x 31 columns]
```

In [12]:
```
test_df.head()
```

Out[12]:
```
    diagnosis  radius_mean  texture_mean  perimeter_mean  area_mean  \
0           1     0.521037      0.022658        0.545989   0.363733
4           1     0.629893      0.156578        0.630986   0.489290
6           1     0.533343      0.347311        0.523875   0.380276
7           1     0.318472      0.376057        0.320710   0.184263
17          1     0.433007      0.370984        0.444406   0.277964

    smoothness_mean  compactness_mean  concavity_mean  concave points_mean  \
0          0.698712          0.792037        0.703799             0.768949
4          0.472434          0.347893        0.464353             0.545217
6          0.401550          0.274891        0.264306             0.386827
7          0.704963          0.445126        0.219653             0.312859
17         0.681210          0.560763        0.403846             0.537376

    symmetry_mean           ...             radius_worst  texture_worst  \
0        0.736186           ...                 0.620776       0.151110
4        0.405742           ...                 0.519744       0.132328
```

```
6        0.397616         ...                   0.531839       0.445077
7        0.615385         ...                   0.324795       0.458736
17       0.598050         ...                   0.463536       0.553785

    perimeter_worst  area_worst  smoothness_worst  compactness_worst  \
0          0.668310    0.450698          0.572692           0.616677
4          0.506948    0.341575          0.397241           0.166732
6          0.511928    0.349194          0.445348           0.218115
7          0.299766    0.174941          0.595331           0.326157
17         0.430251    0.277674          0.690838           0.379982

    concavity_worst  concave points_worst  symmetry_worst  \
0          0.568610              0.912027        0.598462
4          0.319489              0.558419        0.157500
6          0.302236              0.663918        0.295289
7          0.213898              0.534708        0.321506
17         0.382109              0.712371        0.422038

    fractal_dimension_worst
0                  0.418864
4                  0.142595
6                  0.187853
7                  0.393939
17                 0.388036

[5 rows x 31 columns]
```

1.6 Step 5: Build Predictive Model

Using one of the methods (K-Nearest Neighbor, Naïve Bayes, Neural Network, Support Vector Machines, Decision Tree), build your predictive model using the scaled input columns of Training set. You can use any value for the model parameters, or use the default values. In building your model, use k-fold crossvalidation.

> Hint: - http://scikit-learn.org/stable/supervised_learning.html#supervised-learning , - http://scikit-learn.org/stable/modules/cross_validation.html

In [13]:
```python
# Input and Output
inp_train = train_df.iloc[:, 1:]
out_train = train_df["diagnosis"]
inp_test = test_df.iloc[:, 1:]
out_test = test_df["diagnosis"]
```

In [14]:
```python
# Naive Bayes:
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_val_score
nb_model = GaussianNB()
nb_model.fit(inp_train, out_train)
```

```
# Cross validation score of my model
nb_model_scores = cross_val_score(nb_model, inp_train, out_train, cv=10, scoring='accur
print(nb_model_scores)
```

```
[ 0.88372093  0.88372093  0.9047619   0.87804878  0.92682927  0.95121951
  0.90243902  0.97560976  0.92682927  0.95121951]
```

1.7 Step 6. Model Predictions on Training Dataset

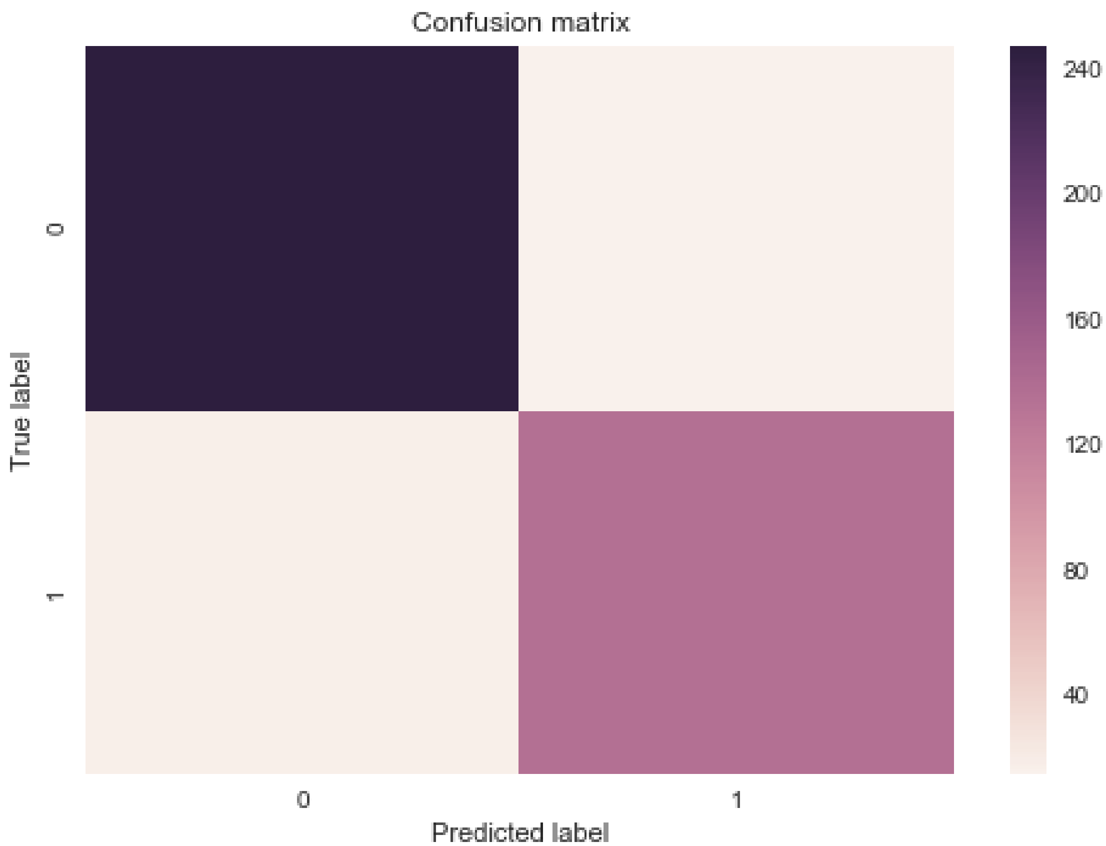
Apply your model to input (scaled) columns of Training dataset to obtain the predicted output for Training dataset. If your model is regression then plot actual output versus predicted output column of Training dataset. If your model is classification then generate confusion matrix on actual and predicted columns of Training dataset.

> Hint: Matplotlip, Seaborn, Bokeh scatter(), plot() functions - http://scikit-learn.org/0.15/auto_examples/plot_confusion_matrix.html - http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html

```
In [15]: # importing libraries for plotting
         # Importing library for confusion matrix
         from sklearn.metrics import confusion_matrix
         import matplotlib.pyplot as plt
         import seaborn as sns
         sns.set(style='darkgrid')
```

```
In [16]: # train prediction for train data
         out_train_pred = nb_model.predict(inp_train)
         # Compute confusion matrix for prediction of train
         cm = confusion_matrix(out_train, out_train_pred)
         print(cm)
         # Show confusion matrix in a separate window
         sns.heatmap(cm)
         plt.title('Confusion matrix')
         plt.ylabel('True label')
         plt.xlabel('Predicted label')
         plt.show()
```

```
[[247  15]
 [ 17 136]]
```



1.8 Step 7. Model Predictions on Test Dataset

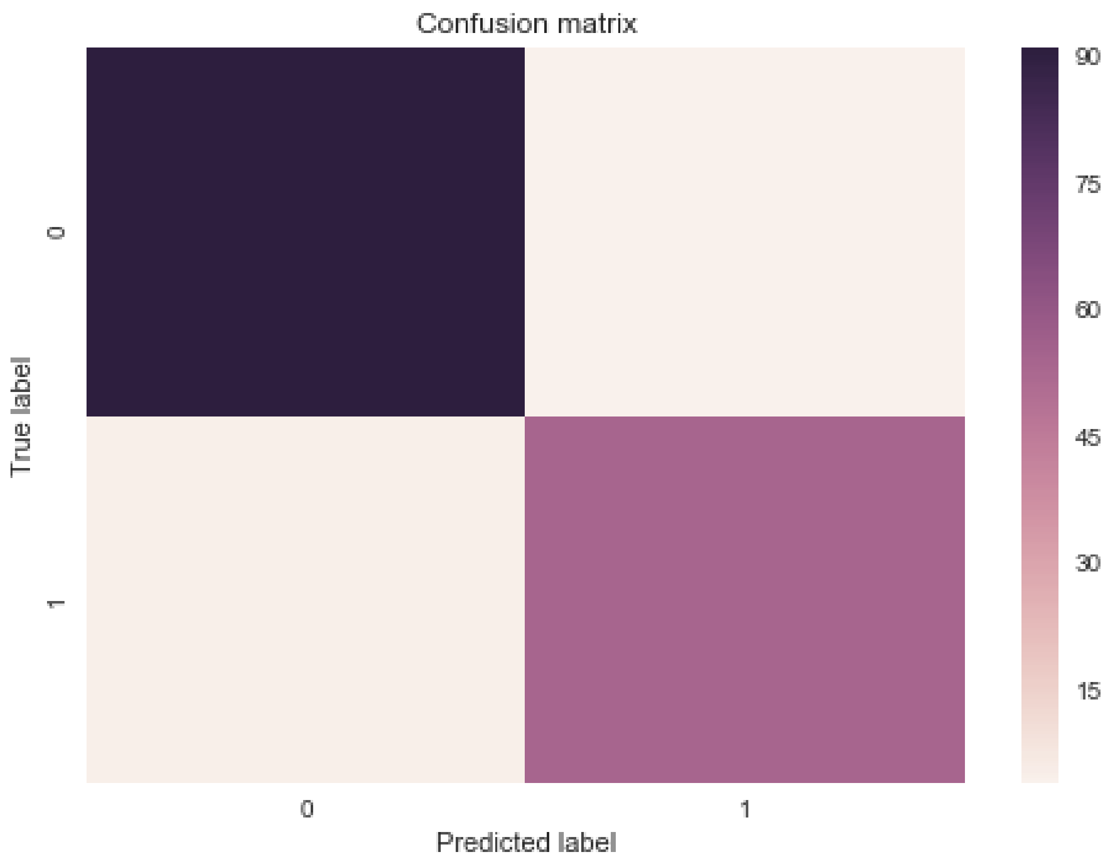
Apply your model to input (scaled) columns of Test dataset to obtain the predicted output for Test dataset. If your model is regression then plot actual output versus predicted output column of Test dataset. If your model is classification then generate confusion matrix on actual and predicted columns of Test dataset.

> Hint: Matplotlip, Seaborn, Bokeh scatter(), plot() functions - http://scikit-learn.org/0.15/auto_examples/plot_confusion_matrix.html - http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html

```
In [17]: # train prediction for train data
         out_test_pred = nb_model.predict(inp_test)
         # Compute confusion matrix for prediction of train
         cm = confusion_matrix(out_test, out_test_pred)
         print(cm)
         # Show confusion matrix in a separate window
         sns.heatmap(cm)
         plt.title('Confusion matrix')
         plt.ylabel('True label')
```

```
plt.xlabel('Predicted label')
plt.show()
```

```
[[91  4]
 [ 5 54]]
```



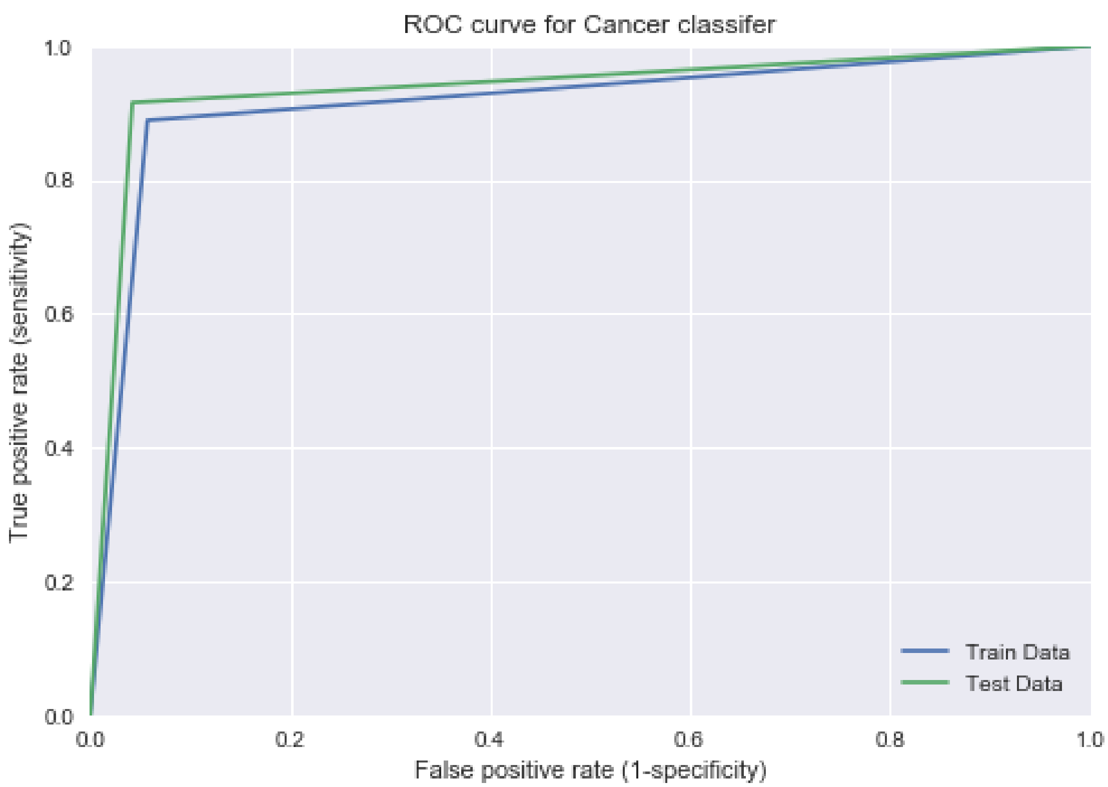
1.9 Step 8. Model Performance

Using one of the error (evaluation) metrics (classification or regression), calculate the performance of the model on Training set and Test set. Compare the performance of the model on Training and Test set. Which one (Training or Testing performance) is better, is there an overfitting case, why ?. Would you deploy (Productionize) this model for using in actual usage in your business system? why ?

Classification Metrics: Accuracy, Precision, Recall, F-score, Recall, AUC, ROC etc Regression Metrics: RMSE, MSE, MAE, R2 etc

- http://scikit-learn.org/stable/model_selection.html#model-selection
- http://scikit-learn.org/stable/modules/model_evaluation.html#classification-report

In [18]:
```
# I would like to use ROC
# Area under ROC Curve (or AUC for short) is
#  a performance metric for binary classification problems.
from sklearn.metrics import roc_curve
# ROC curve for train data
fpr,tpr,thresholds = roc_curve(out_train, out_train_pred)
# plot the curve
plt.plot(fpr, tpr, label="Train Data")
# ROC curve for test data
fpr, tpr, thresholds = roc_curve(out_test, out_test_pred)
# Plotting the curves
plt.plot(fpr, tpr, label="Test Data")
plt.xlim([0.0,1.0])
plt.ylim([0.0,1.0])
plt.title('ROC curve for Cancer classifer')
plt.xlabel('False positive rate (1-specificity)')
plt.ylabel('True positive rate (sensitivity)')
plt.legend(loc=4,)
plt.show()
```



As it seems clear in the plot we created, the Test data is better than the Train data. Which is not expected. **I do not see the traces of overfitting since the test data is also performing well.**

But there is also another chance that Test data is also overfitting... ??

Naive bayes on this particular data set works really good. It might be good for fast prototyping and usage.

1.10 Step 9. Update the Model

Go back to Step5, and choose different values of the model parameters and re-train the model. Repeat Steps: 6 and 7. Using the same error metric, generate the accuracy of the model on Training and Test dataset. Did you get a better performance on Training or Test set? Explain why the new model performs better or worse than the former model.

Let's try to calibrate the GaussianNB(); I will be using isotonic, sigmoid calibration for Gaussian Naive Bayes:

```
In [19]: # For Training Data:
         # Let's remember we have GaussianNB model with
         #  no calibration called out_train_pred

         from sklearn.calibration import CalibratedClassifierCV
         # Gaussian Naive-Bayes with isotonic calibration
         nb_model_isotonic = CalibratedClassifierCV(nb_model, cv=2, method='isotonic')
         nb_model_isotonic.fit(inp_train, out_train)
         out_train_isotonic = nb_model_isotonic.predict_proba(inp_train)[:, 1]
         out_test_isotonic = nb_model_isotonic.predict_proba(inp_test)[:, 1]
```

```
In [20]: # Gaussian Naive-Bayes with sigmoid calibration
         nb_model_sigmoid = CalibratedClassifierCV(nb_model, cv=2, method='sigmoid')
         nb_model_sigmoid.fit(inp_train, out_train)
         out_train_sigmoid = nb_model_sigmoid.predict_proba(inp_train)[:, 1]
         out_test_sigmoid = nb_model_sigmoid.predict_proba(inp_test)[:, 1]
```

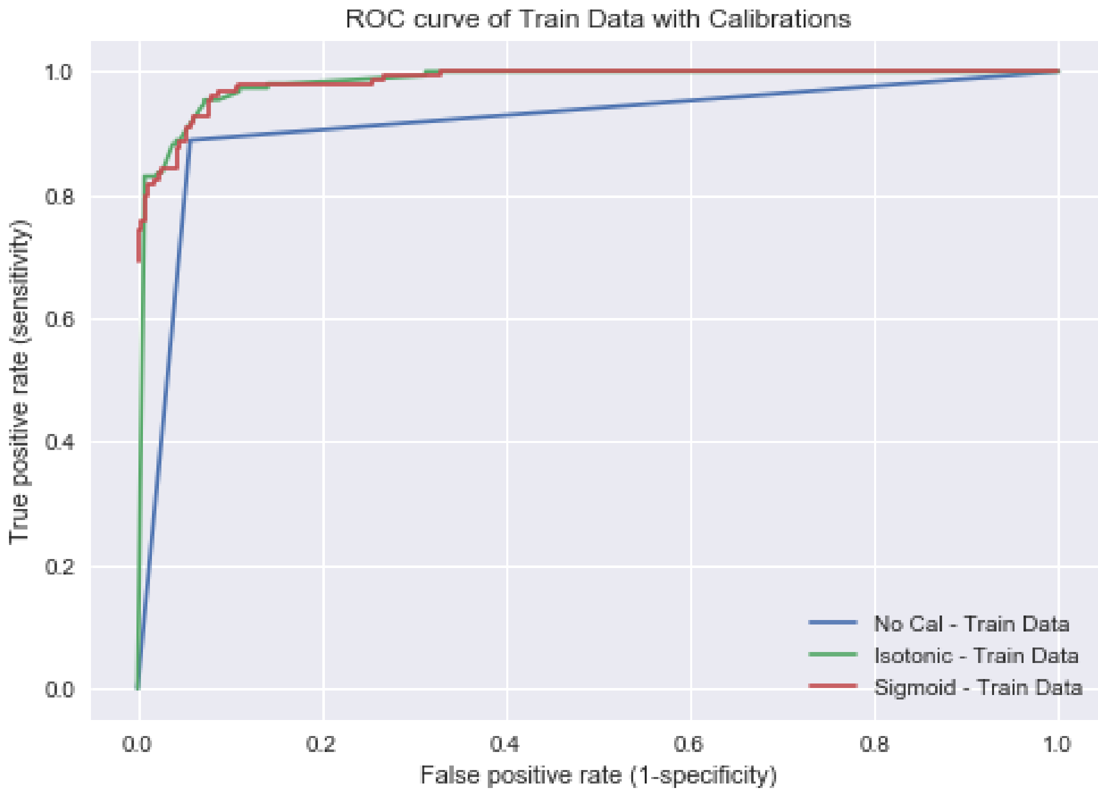
```
In [21]: ## Plotting the comparison of train Data roc_curves
         # ROC curve for train data no calibration
         fpr,tpr,thresholds = roc_curve(out_train, out_train_pred)
         # plot the curve
         plt.plot(fpr, tpr, label="No Cal - Train Data")

         # ROC curve for train data isotonic calibration
         fpr,tpr,thresholds = roc_curve(out_train, out_train_isotonic)
         # plot the curve
         plt.plot(fpr, tpr, label="Isotonic - Train Data")

         # ROC curve for train data sigmoid calibration
         fpr,tpr,thresholds = roc_curve(out_train, out_train_sigmoid)
         # plot the curve
         plt.plot(fpr, tpr, label="Sigmoid - Train Data")

         plt.xlim([-0.05,1.05])
```

```
plt.ylim([-0.05,1.05])
plt.title('ROC curve of Train Data with Calibrations')
plt.xlabel('False positive rate (1-specificity)')
plt.ylabel('True positive rate (sensitivity)')
plt.legend(loc=4,)
plt.show()
```

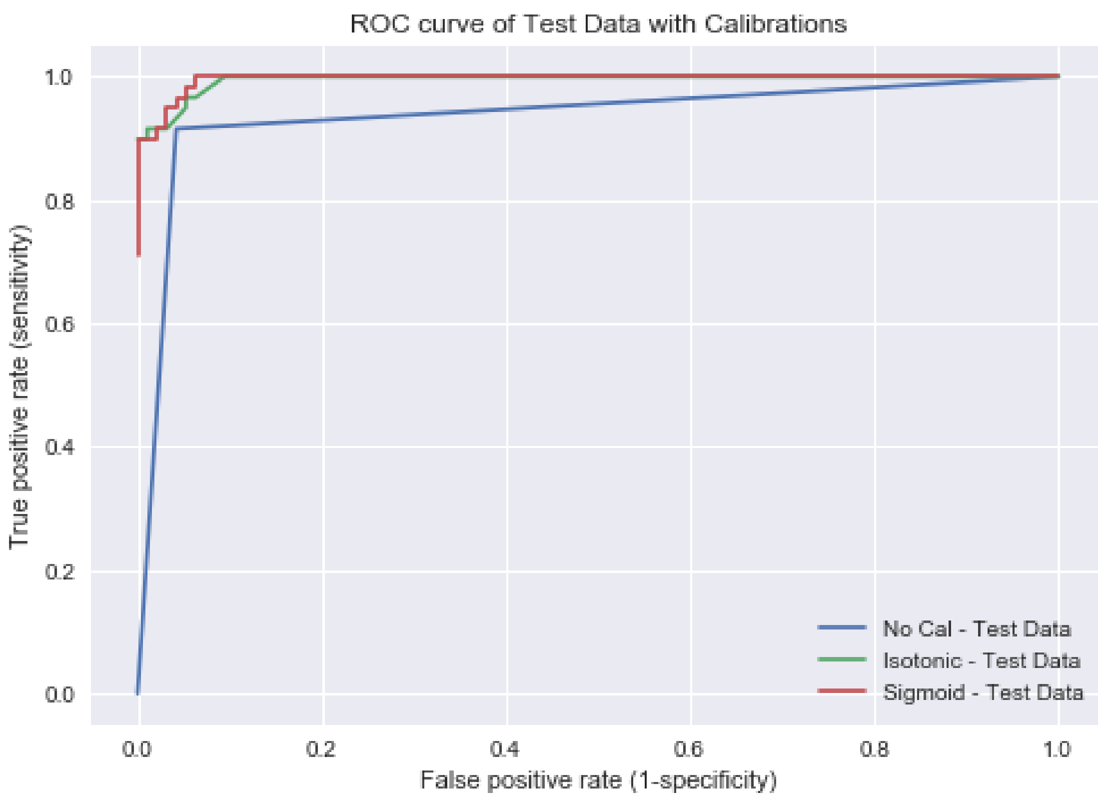


In [22]:
```
# ROC curve for test data no calibration
fpr, tpr, thresholds = roc_curve(out_test, out_test_pred)
# Plotting the curves
plt.plot(fpr, tpr, label="No Cal - Test Data")

# ROC curve for test data isotonic calibration
fpr,tpr,thresholds = roc_curve(out_test, out_test_isotonic)
# plot the curve
plt.plot(fpr, tpr, label="Isotonic - Test Data")

# ROC curve for test data sigmoid calibration
fpr,tpr,thresholds = roc_curve(out_test, out_test_sigmoid)
# plot the curve
plt.plot(fpr, tpr, label="Sigmoid - Test Data")
```

```
plt.xlim([-0.05,1.05])
plt.ylim([-0.05,1.05])
plt.title('ROC curve of Test Data with Calibrations')
plt.xlabel('False positive rate (1-specificity)')
plt.ylabel('True positive rate (sensitivity)')
plt.legend(loc=4,)
plt.show()
```



Extra calibration which add one more layer above the GaussianNB() works better than no calibration. Isotonic and Sigmoid calibrations are performed better than the initial no calibration version.

1.11 Step 10. Change the Error Metric

Choose another error metric other than you used in Step 8 and evaluate the performance of the model on Training and Test dataset by generating the accuracy of the model based on the new metric. Compare the results and explain which error metric is better for your modeling and why?

In [23]:
```
# Checking the error metric to Brier scores
from sklearn.metrics import brier_score_loss

# Checking for only test data predictions
```

```
print("Brier scores: (the smaller the better)")
mdl_score = brier_score_loss(out_test, out_test_pred)
print("No calibration: %1.3f" % mdl_score)
mdl_isotonic_score = brier_score_loss(out_test, out_test_isotonic)
print("With isotonic calibration: %1.3f" % mdl_isotonic_score)
mdl_sigmoid_score = brier_score_loss(out_test, out_test_sigmoid)
print("With sigmoid calibration: %1.3f" % mdl_sigmoid_score)
```

```
Brier scores: (the smaller the better)
No calibration: 0.058
With isotonic calibration: 0.026
With sigmoid calibration: 0.037
```

```
In [24]: # Applying other metrics
         from sklearn import metrics
         print("Printing the different metric results for Not calibrated test data")
         print("-"*60)
         print("Precision score: %1.3f" %
               metrics.precision_score(out_test, out_test_pred))
         print("Recall score on: %1.3f" %
               metrics.recall_score(out_test, out_test_pred))
         print("F1 score on: %1.3f" %
               metrics.f1_score(out_test, out_test_pred) )
         print("Fbeta score with b=0.5 on: %1.3f" %
               metrics.fbeta_score(out_test, out_test_pred, beta=0.5))
         print("Fbeta score with b=1.0 on: %1.3f" %
               metrics.fbeta_score(out_test, out_test_pred, beta=1))
         print("Fbeta score with b=2.0 on: %1.3f" %
               metrics.fbeta_score(out_test, out_test_pred, beta=2))
```

```
Printing the different metric results for Not calibrated test data
------------------------------------------------------------
Precision score: 0.931
Recall score on: 0.915
F1 score on: 0.923
Fbeta score with b=0.5 on: 0.928
Fbeta score with b=1.0 on: 0.923
Fbeta score with b=2.0 on: 0.918
```

When it comes to selecting a way to show how well my models are working I always use both error and accuracy together. In this specific task I had an opportunity to try different metrics available in scikit-learn. In terms of showing a better results, for this model, I would go with Recall score. However I usually go with precision_score.

As the ending remarks for the project I would like to emphasize that Naive Bayes is working suprisingly good for this particular dataset (Breast Cancer from UCI ML website). I am suspecting that my model overfitted because for both test and train data is produced ~92-98% precision, which is quite impossible with ~30 or so features and 500 data points.

I could use more data and selected features to get more real results. For the Final project I am planning to use some techniques that will allow me to select features and only work with them.

*-**Enes K. Ergin**-*