EMORY

GOIZUETA
BUSINESS
SCHOOL

MSBA Team 7 Homework 2b

Janvier Nshimyumukiza, Ashish Sangai, Sherraina Song, Wenqi Zhai

Introduction to Business Analytics

September 16, 2022

Part 1: Classification Models in Python

The main steps our team follows for the KNN and Logistic Regression in Python is as below:

1) EDA (Exploratory Data Analysis)
2) Model Induction
3) Comparison of KNN and Logistic Regression
4) Generalization Performance Metric

I. EDA (Exploratory Data Analysis)

We explored the dataset before building the model by identifying the data size, null values, missing values, and statistical descriptions. On a high-level overview, the dataset contains 32 variables and 569 observations.

Below are the statistical descriptions(summary table) of our data:

```
[ ]  # Describe the data for central tendency and other statistical descriptions
     # for display/output format purpose we turn the descriptions result into a dataframe so that we can see the results for each column
     summary_stats = pd.DataFrame(df.describe())
     print(summary_stats[summary_stats.columns[:15]])
     print(summary_stats[summary_stats.columns[15:]])
```

	0	1	2	3	4
count	5.690000e+02	569.000000	569.000000	569.000000	569.000000
mean	3.037183e+07	0.372583	14.127292	19.289649	91.969033
std	1.250206e+08	0.483918	3.524049	4.301036	24.298981
min	8.670000e+03	0.000000	6.981000	9.710000	43.790000
25%	8.692180e+05	0.000000	11.700000	16.170000	75.170000
50%	9.060240e+05	0.000000	13.370000	18.840000	86.240000
75%	8.813129e+06	1.000000	15.780000	21.800000	104.100000
max	9.113205e+08	1.000000	28.110000	39.280000	188.500000

	5	6	7	8	9
count	569.000000	569.000000	569.000000	569.000000	569.000000
mean	654.889104	0.096360	0.104341	0.088799	0.048919
std	351.914129	0.014064	0.052813	0.079720	0.038803
min	143.500000	0.052630	0.019380	0.000000	0.000000
25%	420.300000	0.086370	0.064920	0.029560	0.020310
50%	551.100000	0.095870	0.092630	0.061540	0.033500
75%	782.700000	0.105300	0.130400	0.130700	0.074000
max	2501.000000	0.163400	0.345400	0.426800	0.201200

	10	11	12	13	14
count	569.000000	569.000000	569.000000	569.000000	569.000000
mean	0.181162	0.062798	0.405172	1.216853	2.866059
std	0.027414	0.007060	0.277313	0.551648	2.021855
min	0.106000	0.049960	0.111500	0.360200	0.757000
25%	0.161900	0.057700	0.232400	0.833900	1.606000
50%	0.179200	0.061540	0.324200	1.108000	2.287000
75%	0.195700	0.066120	0.478900	1.474000	3.357000
max	0.304000	0.097440	2.873000	4.885000	21.980000

	15	16	17	18	19	20
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000
mean	40.337079	0.007041	0.025478	0.031894	0.011796	0.020542
std	45.491006	0.003003	0.017908	0.030186	0.006170	0.008266
min	6.802000	0.001713	0.002252	0.000000	0.000000	0.007882
25%	17.850000	0.005169	0.013080	0.015090	0.007638	0.015160
50%	24.530000	0.006380	0.020450	0.025890	0.010930	0.018730
75%	45.190000	0.008146	0.032450	0.042050	0.014710	0.023480
max	542.200000	0.031130	0.135400	0.396000	0.052790	0.078950

	21	22	23	24	25
count	569.000000	569.000000	569.000000	569.000000	569.000000
mean	0.003795	16.269190	25.677223	107.261213	880.583128
std	0.002646	4.833242	6.146258	33.602542	569.356993
min	0.000895	7.930000	12.020000	50.410000	185.200000
25%	0.002248	13.010000	21.080000	84.110000	515.300000
50%	0.003187	14.970000	25.410000	97.660000	686.500000
75%	0.004558	18.790000	29.720000	125.400000	1084.000000
max	0.029840	36.040000	49.540000	251.200000	4254.000000

	26	27	28	29	30	31
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000
mean	0.132369	0.254265	0.272188	0.114606	0.290076	0.083946
std	0.022832	0.157336	0.208624	0.065732	0.061867	0.018061
min	0.071170	0.027290	0.000000	0.000000	0.156500	0.055040
25%	0.116600	0.147200	0.114500	0.064930	0.250400	0.071460
50%	0.131300	0.211900	0.226700	0.099930	0.282200	0.080040
75%	0.146000	0.339100	0.382900	0.161400	0.317900	0.092080
max	0.222600	1.058000	1.252000	0.291000	0.663800	0.207500

Within the above summary statistics, each column is represented by an integer because the provided dataset does not have the headers.

There are no missing values for any of the columns and the variables we use for prediction are all float numbers.

```
# Reveal the missing values.
df.info()
```

```
RangeIndex: 569 entries, 0 to 568
Data columns (total 32 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       569 non-null    int64  
 1   1       569 non-null    object 
 2   2       569 non-null    float64
 3   3       569 non-null    float64
 4   4       569 non-null    float64
 5   5       569 non-null    float64
 6   6       569 non-null    float64
 7   7       569 non-null    float64
 8   8       569 non-null    float64
 9   9       569 non-null    float64
 10  10      569 non-null    float64
 11  11      569 non-null    float64
 12  12      569 non-null    float64
 13  13      569 non-null    float64
 14  14      569 non-null    float64
 15  15      569 non-null    float64
 16  16      569 non-null    float64
 17  17      569 non-null    float64
 18  18      569 non-null    float64
 19  19      569 non-null    float64
 20  20      569 non-null    float64
 21  21      569 non-null    float64
 22  22      569 non-null    float64
 23  23      569 non-null    float64
 24  24      569 non-null    float64
 25  25      569 non-null    float64
 26  26      569 non-null    float64
 27  27      569 non-null    float64
 28  28      569 non-null    float64
 29  29      569 non-null    float64
 30  30      569 non-null    float64
 31  31      569 non-null    float64
dtypes: float64(30), int64(1), object(1)
memory usage: 142.4+ KB
```

We then created a correlation matrix to understand the multicollinearity between explanatory variables themselves and between target variables and explanatory variables.

```
[ ]  # Correlation Matrix
     plt.figure(figsize = (30,20))

     p1 = round(df.corr(), 2)

     sns.heatmap(p1,cmap="YlGnBu", annot=True)
```

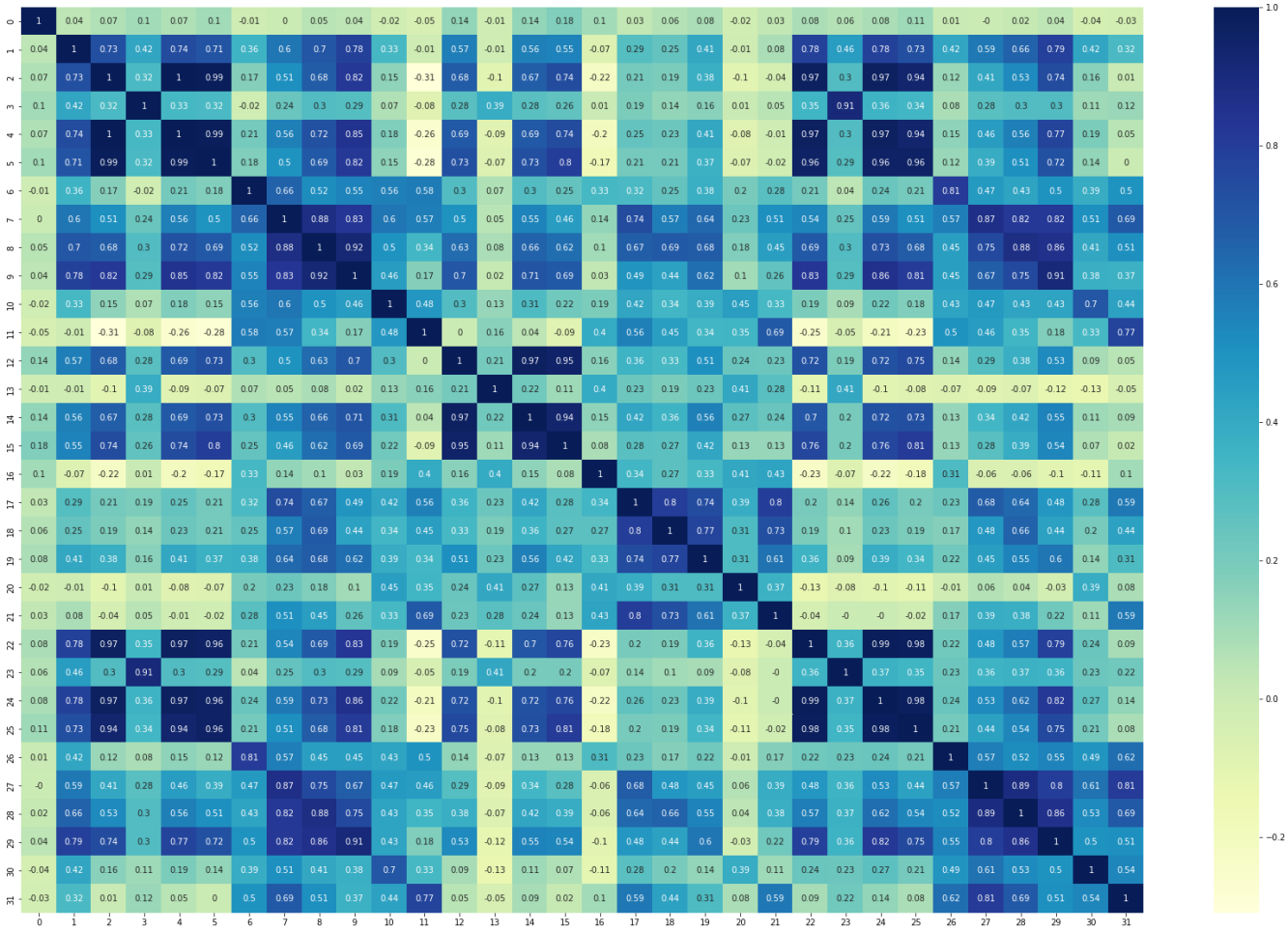
1) Relationship between features and target variable (variable #1):

The type (#1) is strongly correlated with multiple variables (>0.7 corr with 9 variables)

The correlation is relatively high for concave_points_mean (#9), radius_worst (#22), perimeter_worst (#24) and concave_points_worst (#29).

2) Relationship inside the features (all variables except #1):

Variables like smoothness_se (#16) and symmetry_se (#20) are relatively lower correlated to the other variables.



II. Model Induction

1) Preprocessing Steps

a) Label Encoding

We encoded the label from B/M to 0/1 so as to convert them to machine-readable form. The positive class is M (malignant).

```
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
# Encode the labels
le.fit(['B','M'])
# Show the label classes
print(le.classes_)
# Transform the label of the table
df[1] = le.transform(df[1])

df.head(5)
```

```
['B' 'M']
```

	0	1	2	3	4	5	6	7	8	9	...	22	23	24	25	26	27	28	29	30	31
0	842302	1	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	25.38	17.33	184.60	2019.0	0.1622	0.6656	0.7119	0.2654	0.4601	0.11890
1	842517	1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	24.99	23.41	158.80	1956.0	0.1238	0.1866	0.2416	0.1860	0.2750	0.08902
2	84300903	1	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	23.57	25.53	152.50	1709.0	0.1444	0.4245	0.4504	0.2430	0.3613	0.08758
3	84348301	1	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	14.91	26.50	98.87	567.7	0.2098	0.8663	0.6869	0.2575	0.6638	0.17300
4	84358402	1	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	22.54	16.67	152.20	1575.0	0.1374	0.2050	0.4000	0.1625	0.2364	0.07678

5 rows × 32 columns

b) Features Scaling

We normalized the dataset using min-max scaling. KNN relies on majority votes where similarity is modeled by the distance. When the distance calculation is done in KNN, it produces a weak KNN when features are not scaled. When one feature value is larger than the others, that feature will dominate the distance, hence the outcome of the KNN. We decided to use min-max scaling to produce similar range features.

Below we present the python script that we used to preprocess the data.

```
# Normalizing the dataset
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaler.fit(X)
scaled_X = scaler.transform(X)
```

2) Validation Techniques:

```
# Split the data into the training and test data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.40, random_state=42)
```

```
X_train, X_test, y_train, y_test = train_test_split(scaled_X, y, test_size=0.40, random_state=42)
```

We splitted the dataset into training and test sets for both, the original dataset and the normalized version, with a *0.6:0.4* ratio. The KNN and Logistic Classifier models will be built based on the training set and we used the test sets to examine the generalization performance of the two models.

Our team built the two models with both the normalized datasets and the original version. Then, we picked the one with the better performance for each of the models for comparison. The models have exhibited different performances with scaled and unscaled data and it's very interesting as the normalization has the opposite result for the two models.

3) Data Mining:

a) KNN Classifier

To build the model, we used the L2 distance and similarity-moderated 3NN. We set n_neighbor to 3 for three nearest neighbors and we used the metric as 'minkowski' and p = 2 to get the Euclidean

distance. We choose weight as ‘distance’ to apply the similarity-moderated kNN. N_jobs = -1 indicates that we are using all processors for the 3NN classifier.

```
# Set parameters of KNeighborsClassifier
knn = neighbors.KNeighborsClassifier(n_neighbors=3, # n_neighbors is the k in the kNN
                        p=2,                    # power parameter for the Minkowski metric. We used p = 2,
                        metric='minkowski',     # the default metric is minkowski, which is a generalization of the Euclidean distance
                                                   # with p=2 is equivalent to the standard Euclidean distance.
                                                   # with p=1 is equivalent to the Mahattan distance.
                        n_jobs=-1,              # the number of parallel jobs to run for neighbors search. -1 means using all processors
                        weights='distance')      # We choose 'distance' because we wanted to apply a similarity-moderated kNN
```

And we trained the model using the training dataset and made predictions using the model for both in-sample and out-of-sample dataset.

```
knn = train_knn(X_train, y_train)
```

```
# Estimate the predicted values by applying the kNN algorithm
y_pred = knn.predict(X_test)           # make predictions for test set
y_pred_insample = knn.predict(X_train)  # make predictions for train set
```

KNN with unscaled data:

The unscaled KNN yields only a 77% accuracy with an f1-score of 74.17%. However, based on the RapidMiner results we realized that there is room for improvement.

```
Accuracy (out-of-sample): 0.77
Accuracy (in-sample): 1.00
F1 score (out-of-sample):  0.7416993464052287
F1 score (in-sample)    :  1.0
```

KNN with scaled data:

We decided to train the same model on the scaled dataset which yielded a better generalization performance. We trained the same KNN classifier with the scaled training data and we have achieved an out-of-sample accuracy of 96% and F1-score of 95.70% which is pretty good performance in general. Below are the details about the other parameters we used for the model.

```
Accuracy (out-of-sample): 0.96
Accuracy (in-sample): 1.00
F1 score (out-of-sample):  0.9570342762620658
F1 score (in-sample)    :  1.0
```

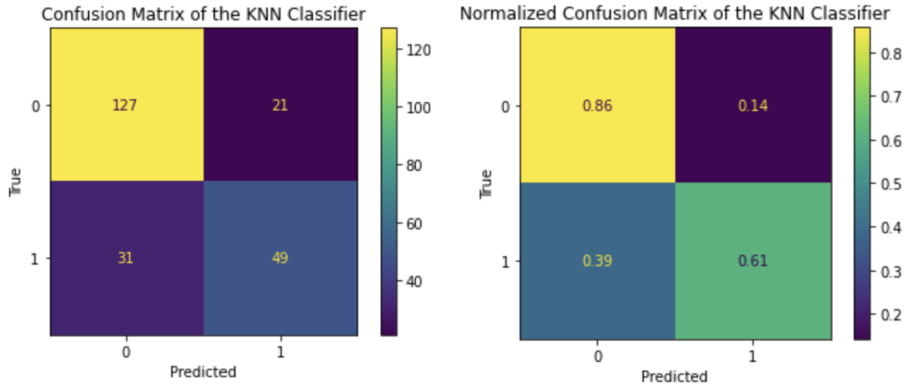
This model has given the above results when built with the scaled dataset. And it’s better compared with the original dataset so we will be using this one for comparison with the Logistic Regression.

	precision	recall	f1-score	support
B	0.98	0.96	0.97	148
M	0.93	0.96	0.94	80
accuracy			0.96	228
macro avg	0.95	0.96	0.96	228
weighted avg	0.96	0.96	0.96	228

Confusion Matrix (Original & Normalized)

```
# Print non normalized confusion matrix
plot_confusion_matrix(list(y_test),
                      list(y_pred),
                      title = 'Confusion Matrix of the KNN Classifier')

# Print normalized confusion matrix
plot_confusion_matrix(list(y_test),
                      list(y_pred),
                      normalize=True,
                      title = 'Normalized Confusion Matrix of the KNN Classifier')
```



b) Logistic Regression

In terms of parameter selection, instead of the default 'lbfgs', we used 'liblinear' solver for the smaller dataset which improved all performance metrics.

We also used the regularization method to avoid overfitting.Firstly, the penalty was changed from L2-norm (using Euclidean distances) to 'L1' because we know that some features are calculated from other features. L1 penalty handles multicollinearity between the features. Secondly, the regularization strength C is set as 1e15. This is a small dataset so we set the solver to be liblinear. We kept all other parameters as default.

```
clf = linear_model.LogisticRegression(multi_class='auto', # accomondates multi-class categorical target variable
                                      penalty='l1', # We used l1 penality for feature selection purpose since some features are calculated from others.
                                      C=1e15, # C parameter is the inverse of regularization strength (i.e., smaller C values
                                             # specify stronger regularization)
                                             # C must be a positive float
                                             # C in this case is 1/lambda
                                             # Applies regularization by default; you can set C very large to avoid regularization
                                             # (setting penalty l2 can speed up the estimations with a very large C)
                                      solver = 'liblinear',   # We used liblinear algorithm in the optimization problem.
                                                              # liblinear solver is recommended by sklearn documentation for small dataset.
                                      max_iter=100)       # we used the default number of max iterations taken for the solvers to converge.
```

We trained the model for both **normalized** and **unnormalized** data and compared their performances respectively.

Fit the data:

```
clf = clf.fit(X_train, y_train)                                    # model induction using the train data
```

Coefficient for the LR mode with unscaled data:

```
clf = clf.fit(X_train, y_train)                          # model induction using the train data
print('The weights of the attributes are:', clf.coef_)      # reports coefficients of the features in the decision function
print('The weights of the intercepts are:', clf.intercept_) # reports intercepts in the decision function
```

```
The weights of the attributes are: [[ 4.50012133e-09 -6.05608356e-01  9.94040853e-02 -9.24863274e-02
  -1.79683456e-03 -4.99625392e+01 -1.44771575e+02  6.34353735e+00
   3.73271714e+02 -7.50807714e+01 -1.66832696e+00  2.39413286e+01
  -1.64451383e+00 -2.07613276e+00  1.85559326e-01 -1.95417798e+02
   7.57587751e+01 -1.33020794e+02  5.41728129e+02 -3.11295954e+02
  -4.12867053e+02 -2.03008077e-01  5.40104235e-01 -5.89954041e-02
   1.03865595e-02 -1.12780065e+01 -1.82353888e+01  4.22661286e+01
   5.74779802e+00  8.43240304e+01 -9.51625883e+00]]
The weights of the intercepts are: [-14.3643838]
```

Coefficient for the LR model with scaled data:

```
The weights of the attributes are: [[   3.96947731  -65.79017769   44.28145705  -61.40906492  109.90793
   -53.8434882  -282.5436425   129.2788061   284.33985389 -105.10280915
   145.20050658  569.94000275  -17.09631305 -252.6640345   161.42005013
    41.00596349  101.07021158 -206.97140419   22.90482459 -140.49460821
  -180.01860035   -7.47998166   57.17147417 -115.80068838  125.83312439
   -67.33817045  -99.34290151  165.39332047  118.96305425  296.71972309
   -85.88332302]]
The weights of the intercepts are: [-101.07158394]
```

Classification Report with scaled data:

	precision	recall	f1-score	support
B	0.98	0.95	0.96	148
M	0.91	0.96	0.93	80
accuracy			0.95	228
macro avg	0.94	0.95	0.95	228
weighted avg	0.95	0.95	0.95	228

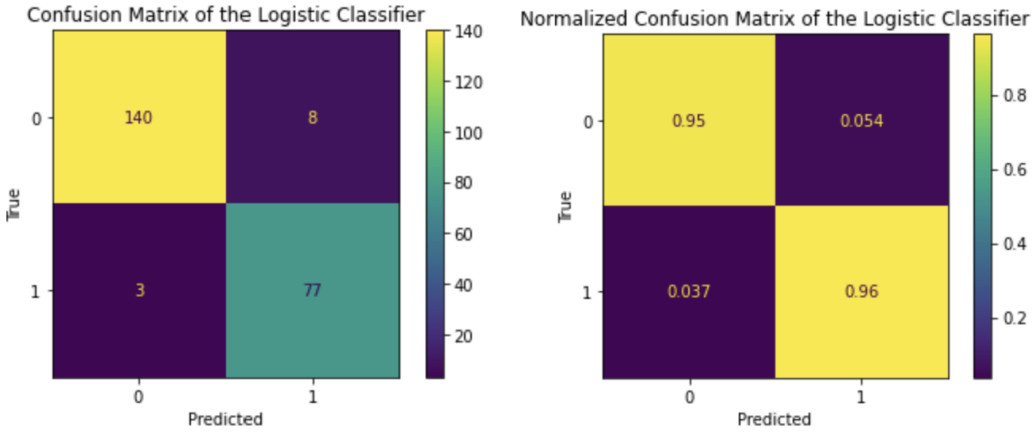
Classification Report with unscaled data:

	precision	recall	f1-score	support
B	0.97	0.94	0.96	148
M	0.89	0.95	0.92	80
accuracy			0.94	228
macro avg	0.93	0.94	0.94	228
weighted avg	0.94	0.94	0.94	228

Confusion Matrix (Original & Normalized)

```
# Print non normalized confusion matrix
plot_confusion_matrix(list(y_test),
                      list(y_pred),
                      title = 'Confusion Matrix of the Logistic Classifier')

# Print normalized confusion matrix
plot_confusion_matrix(list(y_test),
                      list(y_pred),
                      normalize=True,
                      title = 'Normalized Confusion Matrix of the Logistic Classifier')
```



We also train the logistic regression model using the same parameters with the unscaled data and fit the model using the test data. And the generalization performance overall is not as good as using the unscaled data.

III. KNN and Logistic Regression Comparison

In the following section the comparison is based on the best two models: **3NN with normalized data** and **Logistic Regression with unnormalized data**.

We chose the 3NN model with the normalized data because it has shown significantly better performance than 3NN trained on unscaled data. We have previously discussed why we should use KNN trained on scaled/normalized data(in section II).
For logistic regression, we chose the logistic regression trained on original unscaled data because it exhibits a better generalization performance overall. In addition, the regularized logistic regression model with the penalty takes care of overfitting and features' collinearity. Therefore, we decided to keep logistic regression trained on unscaled data as our main logistic classifier.

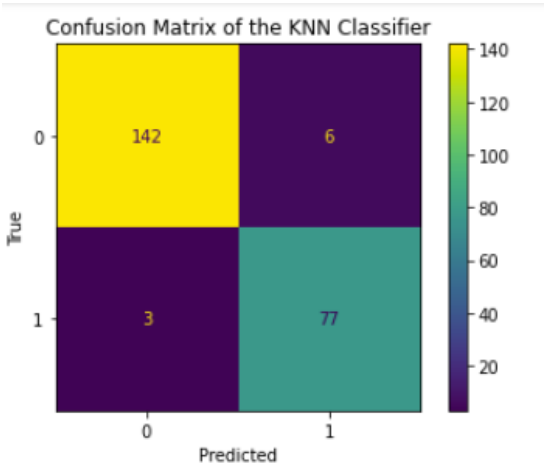
a) Compare the Confusion Matrix (Original & Normalized)

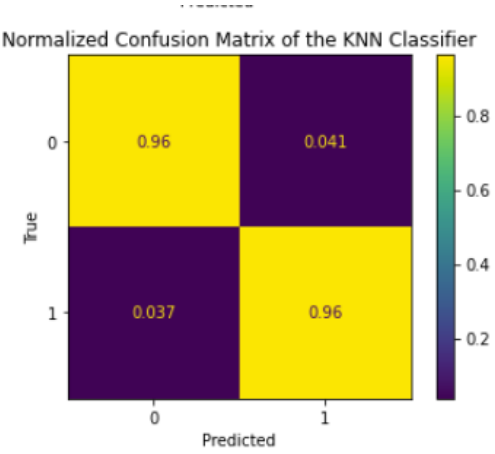
For KNN:

We use the following code to generate the confusion matrix out of the model

```
# Print non normalized confusion matrix
plot_confusion_matrix(list(y_test),
                      list(y_pred),
                      title = 'Confusion Matrix of the KNN Classifier')

# Print normalized confusion matrix
plot_confusion_matrix(list(y_test),
                      list(y_pred),
                      normalize=True,
                      title = 'Normalized Confusion Matrix of the KNN Classifier')
```

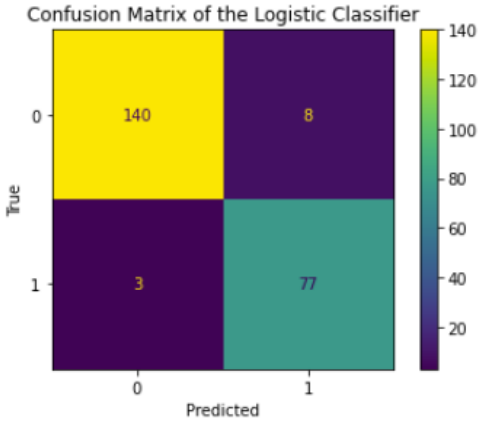


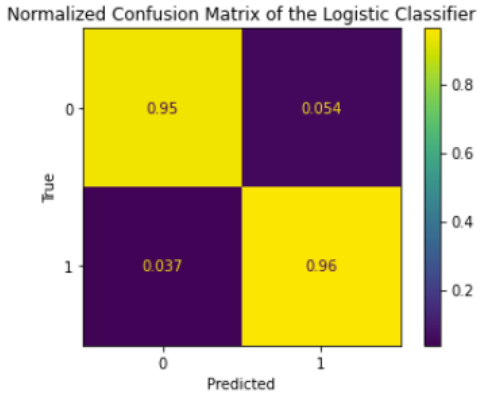


For Logistic Regression:

```
# Print non normalized confusion matrix
plot_confusion_matrix(list(y_test),
                      list(y_pred),
                      title = 'Confusion Matrix of the Logistic Classifier')

# Print normalized confusion matrix
plot_confusion_matrix(list(y_test),
                      list(y_pred),
                      normalize=True,
                      title = 'Normalized Confusion Matrix of the Logistic Classifier')
```





Interpretation:

Our team used the plot_confusion_matrix function to generate both the confusion matrix with the actual figures and the normalized matrix for interpretation. To begin with, false negative error is the most costly error, since we don't want to take the risk of making wrong diagnostics to predict malignant breast tumors as benign. So, in the confusion matrix, we can see that out of 80 malignant samples, both models make three mistakes and yield the same recall of 0.96. These two models have exactly the same performance.

Looking at the first row of the confusion matrix, we can conclude that for the benign class, the KNN model is performing slightly better than the logistic regression model because it makes fewer type I errors to predict the benign tumor as malignant. Out of 148 malignant samples in the test dataset, the KNN makes 6 errors in prediction to wrongly categorize the tumor as benign, while the number for logistic regression is 8. In the real-world context, making a wrong diagnosis of benign tumors is not as serious as making False Negative errors, since we can double-check the result and turn to more advanced methods to exclude the possibility of misdiagnosis, but we still want to minimize both the errors, In a nutshell, based on the confusion matrix, the KNN has overall a better generalization performance in predicting tumors.

b) Compare the accuracy, f1 score, and precision and recall

For KNN:

code:

```
# Accuracy
print('Accuracy (out-of-sample): %.2f' % accuracy_score(y_test, y_pred))
print('Accuracy (in-sample): %.2f' % accuracy_score(y_train, y_pred_insample))

# F1 score
print('F1 score (out-of-sample): ', f1_score(y_test, y_pred, average='macro'))         # average='macro' calculate metrics for each label, and find their unweighted mean
print('F1 score (in-sample)    : ', f1_score(y_train, y_pred_insample, average='macro')) # this macro value does not take label imbalance into account.

# Build a text report showing the main classification metrics (out-of-sample performance)
print(classification_report(y_test, y_pred, target_names=['B','M']))           # builds a text report showing the main classification metrics (precision, recall, f1-score)
```

result:

```
Accuracy (out-of-sample): 0.96
Accuracy (in-sample): 1.00
F1 score (out-of-sample):  0.9570342762620658
F1 score (in-sample)    :  1.0
              precision    recall  f1-score   support

           B       0.98      0.96      0.97       148
           M       0.93      0.96      0.94        80

    accuracy                           0.96       228
   macro avg       0.95      0.96      0.96       228
weighted avg       0.96      0.96      0.96       228
```

For Logistic Regression:

code:

```
# Accuracy
print('Accuracy (out-of-sample): %.2f' % accuracy_score(y_test, y_pred))
print('Accuracy (in-sample): %.2f' % accuracy_score(y_train, y_pred_insample))

# F1 score
print('F1 score (out-of-sample): ', f1_score(y_test, y_pred, average='macro'))         # average='macro' calculate metrics for each label, and find their unweighted mean
print('F1 score (in-sample)    : ', f1_score(y_train, y_pred_insample, average='macro')) # this macro value does not take label imbalance into account.

# Build a text report showing the main classification metrics (out-of-sample performance)
print(classification_report(y_test, y_pred, target_names=['B', 'M'])) # builds a text report showing the main classification metrics (precision, recall, f1-score)
```

result:

```
Accuracy (out-of-sample): 0.95
Accuracy (in-sample): 1.00
F1 score (out-of-sample):  0.947766323024055
F1 score (in-sample)    :  1.0
              precision    recall  f1-score   support

           B       0.98      0.95      0.96       148
           M       0.91      0.96      0.93        80

    accuracy                           0.95       228
   macro avg       0.94      0.95      0.95       228
weighted avg       0.95      0.95      0.95       228
```

Comparing the overall performance, the KNN model yields a better accuracy score of 0.96 vs 0.95 for logistic regression. The F1-measure is also showing a slightly greater performance of 0.9570, ~0.01 better than that of the logistic regression model with 0.9477.

Looking closely at the precision and recall for each group, we can infer that the prediction of KNN for the benign group has the same precision, however the recall is better; while for the malignant group, KNN has higher score for both the precision and recall score, leading to the better F1-score.

IV. Generalization Performance Metric

The generalization performance metric we would use to determine the best performing model, in the context of breast cancer test results, is the **F-measure**. In the medical context of identifying patients whose breast tumor diagnosis is in reality malignant as benign could be a very costly mistake. In this context, it is a binary classification of benign and malignant. It expresses the performance of the machine learning model and gives the combined information about the precision and recall.

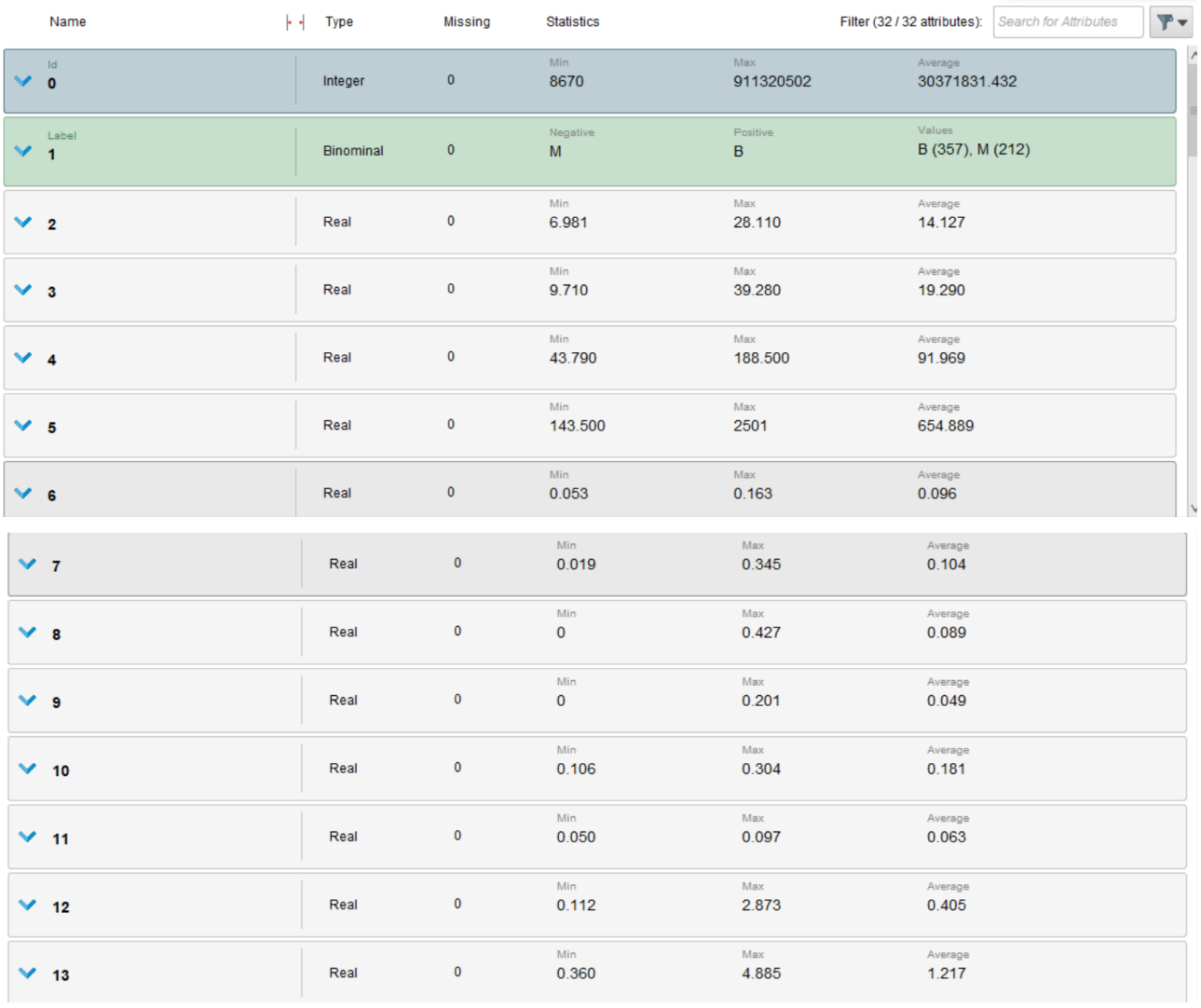
In particular an assumption we made is that the FN rate (misdiagnosis of malignant breast cytology) in this case should be more emphasized. Recall, for the positive class, is an important metric to be considered because it is calculated using TP/(TP+FN). Correspondingly, high recall indicates relatively smaller type II error possibilities.There are high stakes in a patient's life if they receive a wrong diagnosis. If a person receives a false positive, they will be retested further. In comparing the two models, the recall for both, the KNN model and the logistic regression model, is 0.96; indicating similar performance in avoiding making wrong predictions of the actual malignant breast cytology. In the future, we would suggest to further tune the model to make the FN rate even lower, to avoid type two errors.

Part 2: Logistic Regression and KNN Models in Rapidminer

a. Data Preview

Summary statistics of wdbc data in RapidMiner:

We labeled the second column as the label, which is our target variable. The first id column is classified as integer and has the role of id as well. By default the Positive is set as B and because it can't be modified manually we will focus more on the performance of the real positive M class.

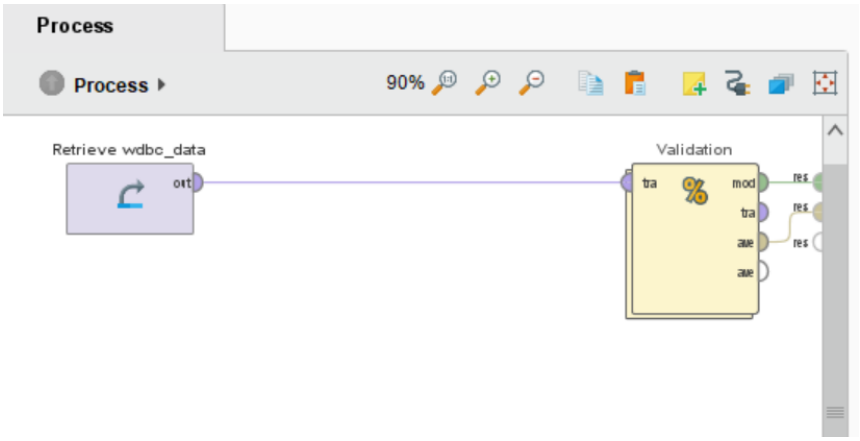


Name	Type	Missing	Statistics		
Filter (32 / 32 attributes): Search for Attributes					
Id 0	Integer	0	Min 8670	Max 911320502	Average 30371831.432
Label 1	Binominal	0	Negative M	Positive B	Values B (357), M (212)
2	Real	0	Min 6.981	Max 28.110	Average 14.127
3	Real	0	Min 9.710	Max 39.280	Average 19.290
4	Real	0	Min 43.790	Max 188.500	Average 91.969
5	Real	0	Min 143.500	Max 2501	Average 654.889
6	Real	0	Min 0.053	Max 0.163	Average 0.096
7	Real	0	Min 0.019	Max 0.345	Average 0.104
8	Real	0	Min 0	Max 0.427	Average 0.089
9	Real	0	Min 0	Max 0.201	Average 0.049
10	Real	0	Min 0.106	Max 0.304	Average 0.181
11	Real	0	Min 0.050	Max 0.097	Average 0.063
12	Real	0	Min 0.112	Max 2.873	Average 0.405
13	Real	0	Min 0.360	Max 4.885	Average 1.217

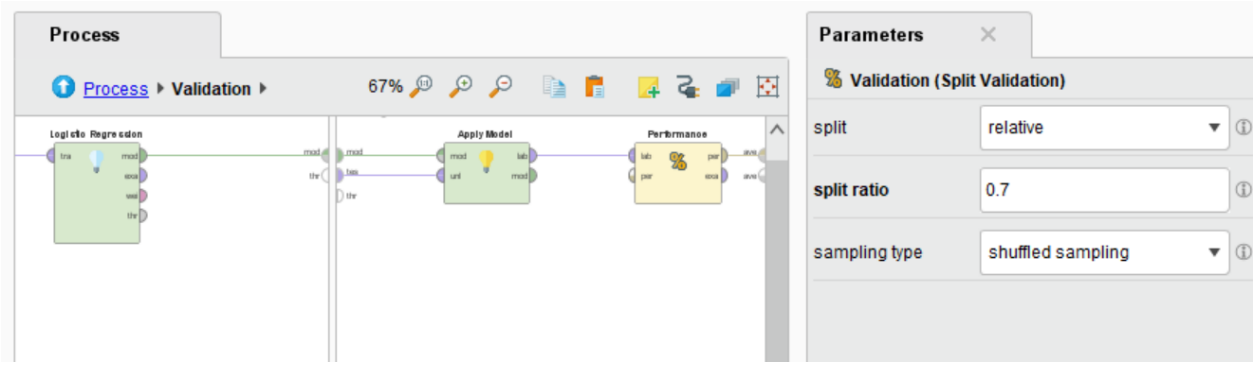
			Min	Max	Average
14	Real	0	0.757	21.980	2.866
15	Real	0	6.802	542.200	40.337
16	Real	0	0.002	0.031	0.007
17	Real	0	0.002	0.135	0.025
18	Real	0	0	0.396	0.032
19	Real	0	0	0.053	0.012
20	Real	0	0.008	0.079	0.021
21	Real	0	0.001	0.030	0.004
22	Real	0	7.930	36.040	16.269
23	Real	0	12.020	49.540	25.677
24	Real	0	50.410	251.200	107.261
25	Real	0	185.200	4254	880.583
26	Real	0	0.071	0.223	0.132
27	Real	0	0.027	1.058	0.254
28	Real	0	0	1.252	0.272
29	Real	0	0	0.291	0.115
30	Real	0	0.157	0.664	0.290
31	Real	0	0.055	0.207	0.084

b. Logistic Regression

i. Logistic Regression Model Setup (Rapidminer Processes)



Upper layer



Inside the Validation

ii. **Logistic Regression Performance**

By default the positive class is set as B but in fact the actual positive class is M. We should be conscious and interpret using the matrix to check the real positive class M's precision and recall.

Confusion matrix and accuracy:

Table View Plot View

accuracy: 95.91%

	true M	true B	class precision
pred. M	72	6	92.31%
pred. B	1	92	98.92%
class recall	98.63%	93.88%	

Precision:

Precision for Class M is 92.31%, for Class B is 98.92%

precision: 98.92% (positive class: B)

	true M	true B	class precision
pred. M	72	6	92.31%
pred. B	1	92	98.92%
class recall	98.63%	93.88%	

Recall:

Recall for Class M is 98.63%, for Class B is 93.88%

recall: 93.88% (positive class: B)

	true M	true B	class precision
pred. M	72	6	92.31%
pred. B	1	92	98.92%
class recall	98.63%	93.88%	

F-measure:

With the Recall and Precision, we can infer that the F-measure for Class M is 95.37%, for Class B is 96.34%

f_measure: 96.34% (positive class: B)

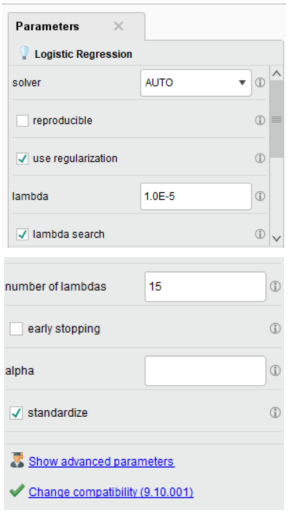
	true M	true B	class precision
pred. M	72	6	92.31%
pred. B	1	92	98.92%
class recall	98.63%	93.88%	

iii. **Logistic Regression Results (Coefficients)**

Attribute	Coefficient	Std. Coefficient
3	-0.057	-0.244
4	0.241	5.864
5	-0.011	-3.881
6	-170.567	-2.399
7	184.947	9.768
8	-103.906	-8.283
9	-124.730	-4.840
10	46.017	1.262
11	-161.858	-1.143
12	-17.595	-4.879
13	3.249	1.792
14	2.214	4.476
15	-0.188	-8.551
16	-336.950	-1.012
17	-262.054	-4.693
18	208.832	6.304
19	-1103.586	-6.809
20	200.797	1.660
21	3697.251	9.783
22	-1.194	-5.769
23	-0.703	-4.320
24	-0.188	-6.306
25	-0.020	-11.449
26	44.470	1.015
27	19.830	3.120
28	-17.328	-3.615
29	3.189	0.210
30	-49.789	-3.080
31	-363.250	-6.561
Intercept	74.435	-3.111

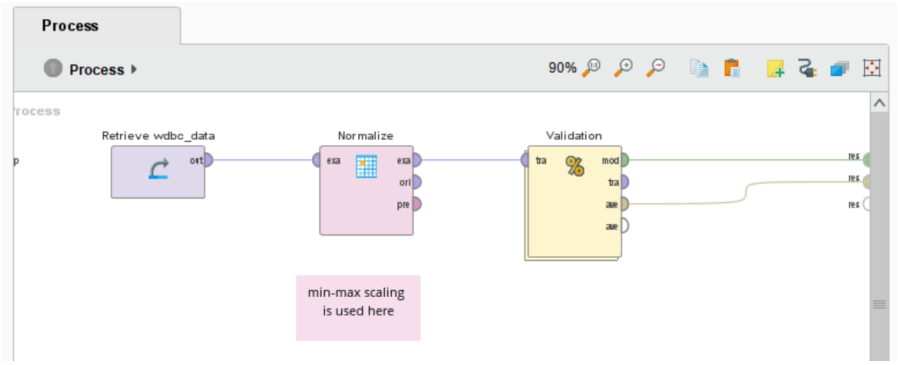
iv. **Logistic Regression Rapidminer Operator Parameters**

We used a regularization method like what we did in Python and set lambda as 1.0E-5 for the model. lambda in this case is 1/C.

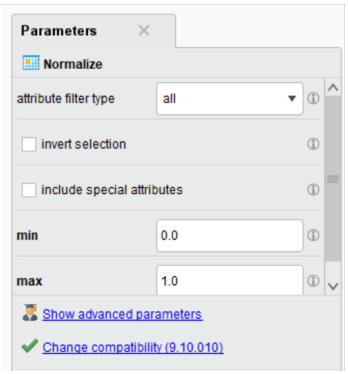


c. kNN

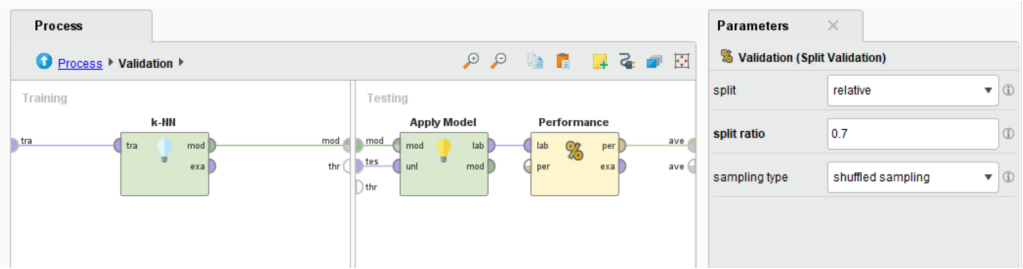
i. kNN Model Setup (Rapidminer Processes)



Upper Layer



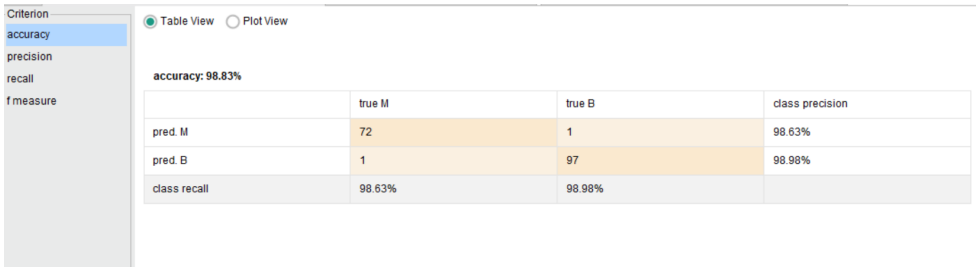
Normalization Using Min-Max Scaling Method



Inside the Validation

ii. **kNN Performance**

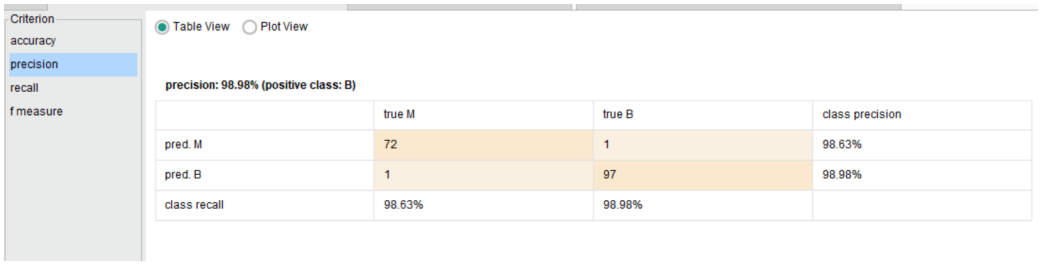
Confusion matrix and accuracy:



	true M	true B	class precision
pred. M	72	1	98.63%
pred. B	1	97	98.98%
class recall	98.63%	98.98%	

Precision:

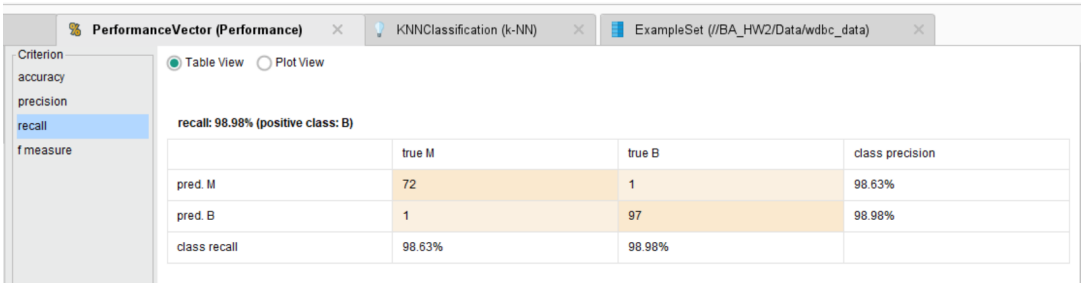
Precision for Class M is 98.63%, for Class B is 98.98%



	true M	true B	class precision
pred. M	72	1	98.63%
pred. B	1	97	98.98%
class recall	98.63%	98.98%	

Recall:

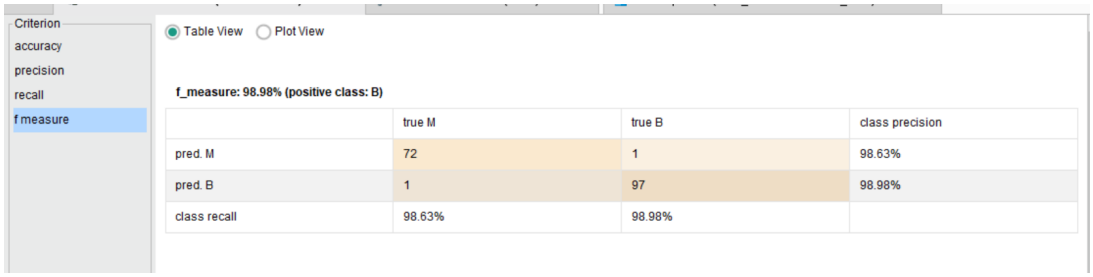
Recall for Class M is 98.63%, for Class B is 98.98%



	true M	true B	class precision
pred. M	72	1	98.63%
pred. B	1	97	98.98%
class recall	98.63%	98.98%	

F-measure:

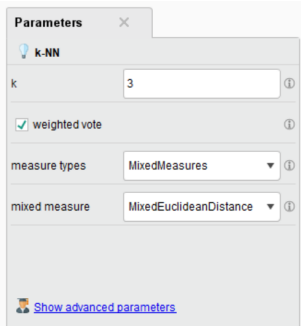
With the Recall and Precision, we can infer that the F-measure for Class M is 98.63%, for class B is 98.98%



	true M	true B	class precision
pred. M	72	1	98.63%
pred. B	1	97	98.98%
class recall	98.63%	98.98%	

iii. **kNN RapidMiner Operator Parameters**

K is set to 3



d. Compare the generalization performance of k-NN with Logistic Regression model

The k-NN model yielded a better f-measure score of 98.63% while the Logistic Regression's f-measure is only around 95.37%. Looking closely, we see the difference is due to the fact that the recall of Class M for the two models is exactly the same at 98.63%, but the precision for Class M is only 92.31%, which indicates that it is making more False positive error to predict the benign breast tumor into malignant ones. And the overall prediction accuracy for kNN is also ~3% higher.

In a nutshell, our models in Rapidminer yielded very similar results and led to consistent conclusions as our kNN and Logistic Regression Models in Python.