CAPAIAN PEMBELAJARAN PRAKTIKUM

1. Menjelaskan konsep dari pembelajaran mesin kemudian di terapkan pada permasalahan – (C2)
2. Mengimplementasikan algoritma dan/atau metode di pembelajaran mesin untuk mendapatkan pemecahan maslaah dan model yang sesuai – (C3)
3. Menghasilkan suatu model terbaik dari permasalahan dan terus melakukan jika terdapat perkembangan di bidang rekayasa cerdas – (C6)

MINGGU 14

EVALUATING TECHNIQUES

DESKRIPSI TEMA

Mahasiswa mempelajari cara menghitung akurasi model model dengan teknik evaluasi model sehingga model bisa tervalidasi dengan baik dan optimal

CAPAIAN PEMBELAJARAN MINGGUAN (SUB CPMK)

Mahasiswa dapat memahami dan menghitung akurasi model dengan teknik evaluasi model sehingga menghasilkan model pembelajaran yang optimal dengan Bahasa pemrogramam Python

PERALATAN YANG DIGUNAKAN

Anaconda Python 3
Laptop atau Personal Computer

LANGKAH-LANGKAH PRAKTIKUM

In previous week, we learn about how to build a model. Let's we build a model in this week before we entered the evaluating model.

CLASSIFICATION MODEL

In all classification (or supervised learning) problems, the first step after preparing the whole dataset is to segregate the data into testing and training set and optionally validation set. For our classification example, we will use a popular multi-class classification problem, handwritten digit recognition. The data for the same is available as part of *scikit-learn* library. The problem here is to predict the actual from a handwritten image of a digit. In its original from the problem comes in the domain of image based classification and computer vision. In the dataset we have is a 1x64 feature vector, which represent the image representation of grey scale image of the handwritten digit.

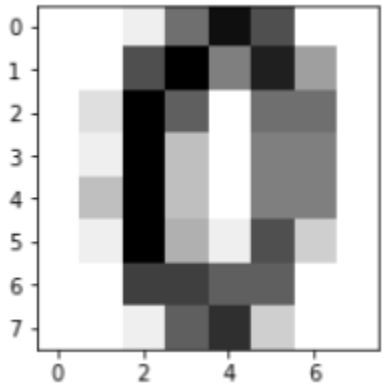
1. Before we proceed to build, let's first see how both the data and the image we intend to analyze look. The following code will load the data for the image at index 10 and plot it :

```
1  from sklearn import datasets, metrics
2  import matplotlib.pyplot as plt
3
4  %matplotlib inline
```

```
1 digits = datasets.load_digits()
```

```
1 plt.figure(figsize=(3,3))
2 plt.imshow(digits.images[10],cmap=plt.cm.gray_r)
```

The image generated by the code is depicted and representing the digit zero.



2. We can determine how the raw pixel data looks the flattened vector representation and the number (class label) , which represented by the image using the following code :

Actual Image :

```
1 digits.images[10]
```

```
array([[ 0.,  0.,  1.,  9., 15., 11.,  0.,  0.],
       [ 0.,  0., 11., 16.,  8., 14.,  6.,  0.],
       [ 0.,  2., 16., 10.,  0.,  9.,  9.,  0.],
       [ 0.,  1., 16.,  4.,  0.,  8.,  8.,  0.],
       [ 0.,  4., 16.,  4.,  0.,  8.,  8.,  0.],
       [ 0.,  1., 16.,  5.,  1., 11.,  3.,  0.],
       [ 0.,  0., 12., 12., 10., 10.,  0.,  0.],
       [ 0.,  0.,  1., 10., 13.,  3.,  0.,  0.]])
```

Flattened Image :

```
1 digits.data[10]
```

```
array([ 0.,  0.,  1.,  9., 15., 11.,  0.,  0.,  0.,  0., 11., 16.,  8.,
       14.,  6.,  0.,  0.,  2., 16., 10.,  0.,  9.,  9.,  0.,  0.,  1.,
       16.,  4.,  0.,  8.,  8.,  0.,  0.,  4., 16.,  4.,  0.,  8.,  8.,
        0.,  0.,  1., 16.,  5.,  1., 11.,  3.,  0.,  0.,  0., 12., 12.,
       10., 10.,  0.,  0.,  0.,  0.,  1., 10., 13.,  3.,  0.,  0.])
```

Image class label :

```
digits.target[10]
```

```
0
```

We will later see that we can frame this problem in a variety of ways. But for this, we will use a logistic regression model to do this classification.

3. We split the dataset into separate test and train set. The size of test set is generally dependent on the total amount of data available. In our example, we will use a test set, which is 30% of the overall dataset. The total data points in each dataset is printed for ease of understanding

```
X_digits = digits.data
y_digits = digits.target

num_data_points = len(X_digits)
X_train = X_digits[:int(.7*num_data_points)]
y_train = y_digits[:int(.7*num_data_points)]
X_test = X_digits[:int(.7*num_data_points)]
y_test = y_digits[:int(.7*num_data_points)]
print(X_train.shape,X_test.shape)
```

```
(1257, 64) (1257, 64)
```

From the preceding output, we can see our training dataset has 1257 data points and the test set has 540 data points.

4. The next step is specifying the model that we will be using and the hyperparameter values that we want to use. The values of hyperparameter do not depend on the underlying data and are usually set prior to model training and are fine tuned for extracting the best model.

```
from sklearn import linear_model
logistic = linear_model.LogisticRegression()
logistic.fit(X_train,y_train)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

5. Let's now test the accuracy of this model on the test dataset.

```
print('Logistic Regression mean Accuracy : %f' %logistic.score(X_test,y_test))
```

```
Logistic Regression mean Accuracy : 0.964912
```

This concludes our very basic example of fitting a classification model on our dataset. Note that our dataset was in a fully processed and cleaned format. We need to ensure our data is in the same way before we proceed to fit any models on it when solving any problem.

EVALUATING THE CLUSTERING

In this section, we will learn how we can fit a clustering model on another dataset. In the example which we will pick, we will use a labeled dataset to help us see the result of the clustering model and compare it with actual label. A point to remember here is that, usually labeled data is not available in the real world, which is why we choose to go for unsupervised methods like clustering. We will try to cover two different algorithms, one each from partitioning based clustering and hierarchical clustering.

The data that we will use for our clustering example will be very popular Wisconsin Diagnostic Breast Cancer dataset. This dataset has 30 attributes and corresponding label for each data point (breast mass) depicting if it has cancer (malignant : label value 0) or no cancer (benign : label value 1).

6. Let's load the data using the following code

```
#Using Wisconsin Breast Cancer dataset
import numpy as np
from sklearn.datasets import load_breast_cancer

#load data
data = load_breast_cancer()
X = data.data
y= data.target
print(X.shape,data.feature_names)
```

```
(569, 30) ['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

It is evident that we have a total 569 observations and 30 attributes or features for each observation.

Partitioning Based Clustering

We will choose the simplest yet popular partitioning based clustering model for our example, which is K-Means algorithm. This algorithm is a centroid based clustering algorithm, which starts with some assumption about the total clusters in the data and with random centers assigned to each of the clusters. It then reassigns each data point to the center closest to it, using Euclidean distance as the distance metric. Variants include algorithms like K-Medoids.

Since we already known from the data labels that we have two possible types of categories either 0 or 1, the following code tries to determine these two clusters from the data by leveraging K-Means clustering. In the real world, this is not always the case since we will not know the possible number of clusters. This is one of the most important downsides of K-Means clustering.

7. Import the KMeans :

```
#Unsupervised learning-partitioning
from sklearn.cluster import KMeans

km = KMeans(n_clusters=2,random_state=2)
km.fit(X)

labels = km.labels_
centers=km.cluster_centers_
print(labels[:10])
```

```
[0 0 0 1 0 1 0 1 1 1]
```

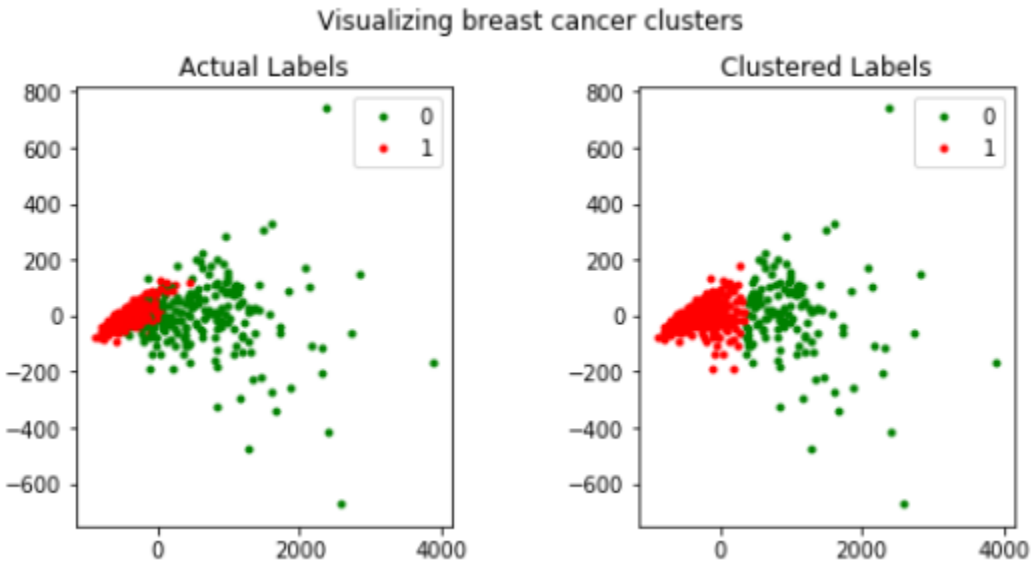
Once the fit process is complete we can get the centers and labels of our two clusters in the dataset by using the preceding attributes. The centers here refer to some numerical value of the dimension of the data (the 30 attributes in the dataset) around which data in clustered.

8. Because we dealing with 30 features and visualizing the centers on a 30-dimensional features space would be impossible to interpret or even perform. Hence, we will leverage PCA to reduce the input dimensions to two principal components and visualize the clusters on top of the same.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
bc_pca = pca.fit_transform(X)
```

9. The following code helps visualize the clusters on the reduced 2D feature space for the actual labels as well as the clustered output labels

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
fig.suptitle('Visualizing breast cancer clusters')
fig.subplots_adjust(top=0.85, wspace=0.5)
ax1.set_title('Actual Labels')
ax2.set_title('Clustered Labels')

for i in range(len(y)):
    if y[i] == 0:
        c1 = ax1.scatter(bc_pca[i,0], bc_pca[i,1],c='g', marker='.')
    if y[i] == 1:
        c2 = ax1.scatter(bc_pca[i,0], bc_pca[i,1],c='r', marker='.')

    if labels[i] == 0:
        c3 = ax2.scatter(bc_pca[i,0], bc_pca[i,1],c='g', marker='.')
    if labels[i] == 1:
        c4 = ax2.scatter(bc_pca[i,0], bc_pca[i,1],c='r', marker='.')

l1 = ax1.legend([c1, c2], ['0', '1'])
l2 = ax2.legend([c3, c4], ['0', '1'])
```



From the figure, we can clearly see that the clustering has worked quite well and it shows distinct separation between clusters with labels 0 and 1 and is quite similar to the actual labels. However, we do have some overlap where we have mislabeled some instances, which is evident in the plot on the right.

Hierarchical Clustering

We can use the same data to perform hierarchical clustering and see if the result much as compared to K Means clustering and the actual class. In *scikit-learn* we have a multitude of interfaces like the ***AgglomerativeClustering*** class to perform hierarchical clustering.

We will leverage low-level function from scipy however because we still need to mention the number of clusters in the AgglomerativeClustering interface which we want to avoid. Since we already have the breast cancer feature set in variable X, the following code help us compute the linkage matrix using Ward's minimum variance criterion.

10. Using AgglomerativeClustering and Ward's distance metric :

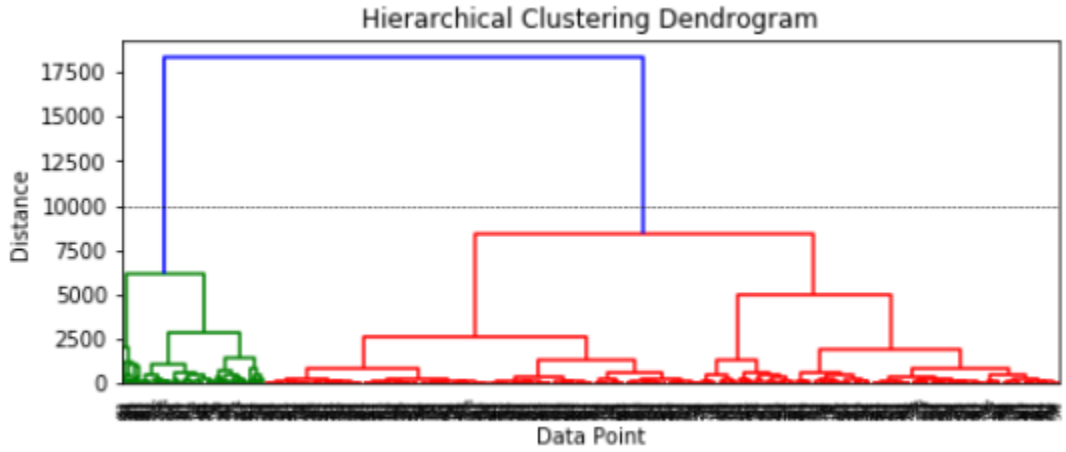
```
#Unsupervised Leaning - Hierachical

from scipy.cluster.hierarchy import dendrogram,linkage
import numpy as np
np.set_printoptions(suppress= True)

Z=linkage(X,'ward')
print(Z)
```

```
[[  287.           336.             3.81596727     2.        ]
 [  106.           420.             4.11664267     2.        ]
 [   55.           251.             4.93361024     2.        ]
 ...
 [ 1130.          1132.          6196.07482529    86.        ]
 [ 1131.          1133.          8368.99225244   483.        ]
 [ 1134.          1135.         18371.10293626   569.        ]]
```

11. Visualize these distance – based merges is to use a dendrogram :

```
plt.figure(figsize=(8,3))
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Data Point')
plt.ylabel('Distance')
dendrogram(Z)
plt.axhline(y=10000, c='k',ls='--',lw=0.5)
plt.show()
```

In the dendrogram, we can see how each data point starts as an individual cluster and slowly starts getting merged with other data points to form clusters. On a high level from the colors and dendrogram, we can see that the model has correctly identified two major clusters if we consider a distance metric of around 10 000 or above.



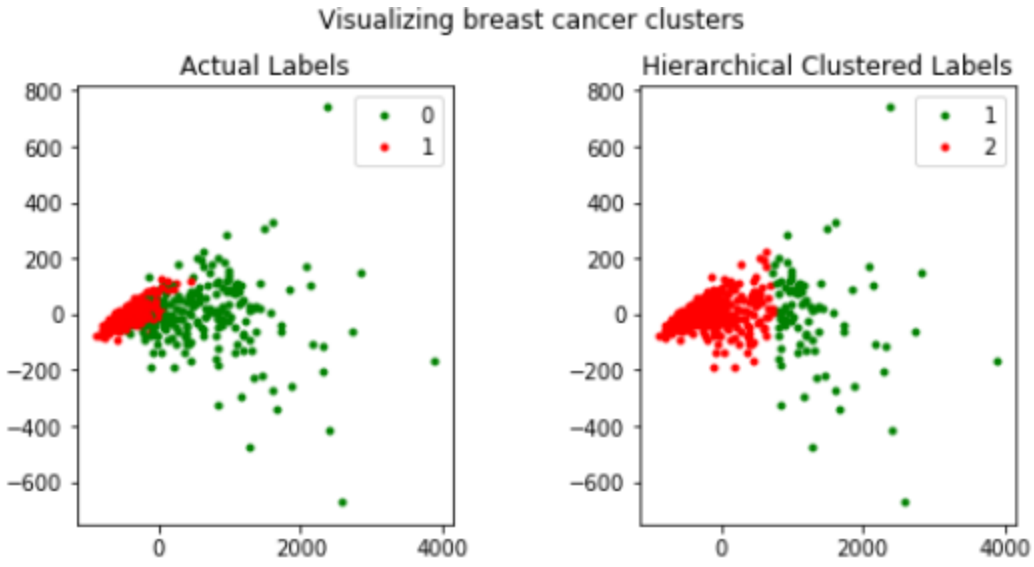
12. Leverage this distance, we can get the cluster labels using following code :

```
from scipy.cluster.hierarchy import fcluster
max_dist = 10000
hc_labels = fcluster(Z,max_dist, criterion ='distance')
```

13. Compare how the cluster outputs look based on the PCA reduced dimensions as compared to the original label distribution.

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
fig.suptitle('Visualizing breast cancer clusters')
fig.subplots_adjust(top=0.85, wspace=0.5)
ax1.set_title('Actual Labels')
ax2.set_title('Hierarchical Clustered Labels')

for i in range(len(y)):
    if y[i] == 0:
        c1 = ax1.scatter(bc_pca[i,0], bc_pca[i,1],c='g', marker='.')
    if y[i] == 1:
        c2 = ax1.scatter(bc_pca[i,0], bc_pca[i,1],c='r', marker='.')

    if hc_labels[i] == 1:
        c3 = ax2.scatter(bc_pca[i,0], bc_pca[i,1],c='g', marker='.')
    if hc_labels[i] == 2:
        c4 = ax2.scatter(bc_pca[i,0], bc_pca[i,1],c='r', marker='.')

l1 = ax1.legend([c1, c2], ['0', '1'])
l2 = ax2.legend([c3, c4], ['1', '2'])
```

We definitely see two distinct clusters but there is more overlap as compared to the K Means method between the 2 clusters and we have more mislabeled instances. The advantage of this method is that we don't need to input the number of clusters beforehand and model tries to find it from underlying data.



MODEL EVALUATION

We have seen the process of data retrieval, processing, wrangling and modeling based on various requirements. A logical question that follows is how we can make the judgement whether a model is good or bad? Just because we have developed something fancy using renowned algorithm, doesn't guarantee its performance will be great. Model evaluation is the answer to these questions and is an essential part of the whole Machine Learning pipeline.

So how we evaluate a model? How can we make a decision whether Model A is better or Model B performs better? The ideal way is to have some numerical measure or metric of a model's effectiveness and use that measure to rank and select models.

Evaluating Classification Models

14. Prepare train and test datasets to build our classification models. We will be leveraging by X and y variables from before, which holds the data and labels for the breast cancer dataset observations

```
from sklearn.model_selection import train_test_split

X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3, random_state=42)
print(X_train.shape,X_test.shape)
```

```
(398, 30) (171, 30)
```

From the preceding output, it is clear that we have 398 observations in our train dataset and 171 observations in our test dataset. We will be leveraging a nifty module we have created for model evaluation, name ***model_evaluation_utils*** (attach in your e-learning).

Confusion Matrix

Confusion matrix is one of the most popular ways to evaluate a classification model. Although the matrix by itself is not a metric, the matrix representation can be used to define a variety of metrics, all of which become important in some specific case or scenario. A confusion matrix can be created for a binary classification as well as multi-class classification model.

15. Let's build a logistic regression model on our breast cancer dataset and look at the confusion matrix for the model predictions on the test dataset.

```
from sklearn import linear_model

logistic=linear_model.LogisticRegression()
logistic.fit(X_train,y_train)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)
```

16. The preceding output depicts the confusion matrix with necessary annotation we can see that out of 63 observations with label 0 (malignant), our model correctly predicted 59 observations. Similarly out of 108 observations with label 1 (benign), our model has correctly predicted 106 observation. More detailed analysis is coming right in below information of our model predictions on the breast cancer test data.

```
#confusion matrix
import model_evaluation_utils as meu

y_pred = logistic.predict(X_test)
meu.display_confusion_matrix(true_labels=y_test, predicted_labels=y_pred, classes=[0, 1])
```

```
           Predicted:
                     0    1
Actual: 0           59    4
        1            2  106
```

```
possitive_class = 1
TP = 106
FP=4
TN=9
FN=2
```

Performance Metrics

17. Accuracy : this is one of the most popular measures of classifier performance, it is defined as the overall accuracy or proportions of correct predictions of the model. The following code computes accuracy on our model predictions :

```
fw_acc = round(meu.metrics.accuracy_score(y_true=y_test, y_pred=y_pred), 5)
mc_acc = round((TP + TN) / (TP + TN + FP + FN), 5)
print('Framework Accuracy:', fw_acc)
print('Manually Computed Accuracy:', mc_acc)
```

```
Framework Accuracy: 0.96491
Manually Computed Accuracy: 0.95041
```

18. Precision : also known as positive value, is another metric that can be derived from the confusion matrix. It is defined as the number of predictions made that are actually correct or relevant out of all predictions based in the positive class. The following code computes precision on our model predictions :

```
fw_prec = round(meu.metrics.precision_score(y_true=y_test, y_pred=y_pred), 5)
mc_prec = round((TP) / (TP + FP), 5)
print('Framework Precision:', fw_prec)
print('Manually Computed Precision:', mc_prec)
```

```
Framework Precision: 0.96364
Manually Computed Precision: 0.96364
```

19. Recall : also known as sensitivity, is a measure of a model to identify the percentage of relevant data points. it is defined as the number of instance of the positive class that were correctly predicted. The following code computes recall on our model predictions :

```
fw_rec = round(meu.metrics.recall_score(y_true=y_test, y_pred=y_pred), 5)
mc_rec = round((TP) / (TP + FN), 5)
print('Framework Recall:', fw_rec)
print('Manually Computed Recall:', mc_rec)
```

```
Framework Recall: 0.98148
Manually Computed Recall: 0.98148
```

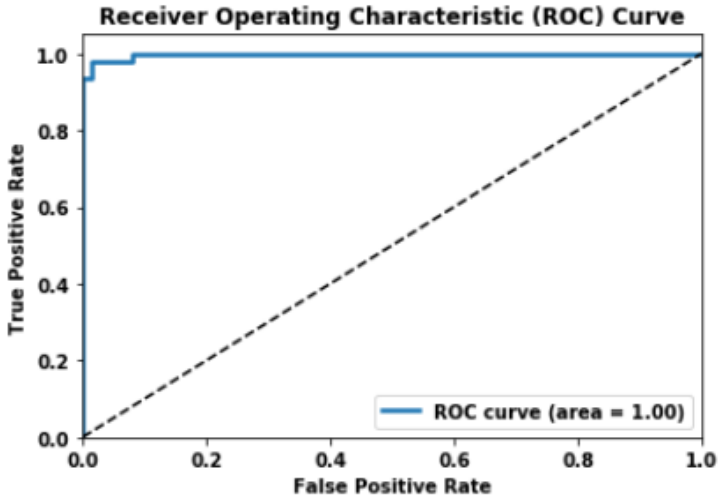
20. F1 – Score : there are some cases in which we want a balanced optimization of both precision and recall. F1 score is metric that is the harmonic mean of precision and recall and help us optimize a classifier for balanced precision and recall performance. Let's compute the F1 score on the predictions made by our mode using the following code :

```
fw_f1 = round(meu.metrics.f1_score(y_true=y_test, y_pred=y_pred), 5)
mc_f1 = round((2*mc_prec*mc_rec) / (mc_prec+mc_rec), 5)
print('Framework F1-Score:', fw_f1)
print('Manually Computed F1-Score:', mc_f1)
```

```
Framework F1-Score: 0.97248
Manually Computed F1-Score: 0.97248
```

21. Receiver Operating Characteristic Curve (ROC Curve) : the curve can be created by plotting the fraction of true positive versus the fraction of false positive. It is applicable mostly for scoring classifiers. Scoring classifiers are the type of classifiers which will return a probability value or score for each class label, from which a class label can be deduced (based on maximum probability value). This curve can be plotted using the True Positive Rate (TPR) and False Positive Rate (FPR) of a classifier. TPR is known as sensitivity or recall. FPR is known as false alarms or (1 – specificity), determining the total number of incorrect positive predictions among all negative samples in the dataset.
    The following code plots the ROC curve for our breast cancer logistic regression model leveraging the same function (model_evaluation_utils).

```
meu.plot_model_roc_curve(clf=logistic, features=X_test, true_labels=y_test)
```



Evaluating Clustering Models

22. To illustrate the evaluation metrics with a real world example, we will leverage the breast cancer dataset available in the variables X for the data y for the observation labels. We will also use the K means algorithm to fit two model on this data – one with two clusters and the second one with five clusters – and then evaluate their performance

```
km2 = KMeans(n_clusters=2, random_state=42).fit(X)
km2_labels = km2.labels_

km5 = KMeans(n_clusters=5, random_state=42).fit(X)
km5_labels = km5.labels_
```

External Validation

External validation means validating the clustering model when we have some ground truth available as labeled data. The presence of external labels reduces most of the complexity of model evaluation as the clustering (unsupervised) model can be validated in similar fashion to classification models. Three popular metrics can be used in this scenario.

- Homogeneity : a clustering model prediction result satisfies homogeneity if all of its clusters contain only data points that are members of a single class (based on the true class labels)
- Completeness : a clustering model predictions result satisfies completeness if all the data points of a specific ground truth class label are also elements of the same cluster
- V measure: the harmonic mean of homogeneity and completeness score gives us the V measure value

23. These values of external validation are typically bounded between 0 and 1 and usually higher values are better. Let's compute these metric on our K Means clustering model :

```
km2_hcv = np.round(metrics.homogeneity_completeness_v_measure(y, km2_labels), 3)
km5_hcv = np.round(metrics.homogeneity_completeness_v_measure(y, km5_labels), 3)

print('Homogeneity, Completeness, V-measure metrics for num clusters=2: ', km2_hcv)
print('Homogeneity, Completeness, V-measure metrics for num clusters=5: ', km5_hcv)

```

```
Homogeneity, Completeness, V-measure metrics for num clusters=2:  [0.422 0.517 0.465]
Homogeneity, Completeness, V-measure metrics for num clusters=5:  [0.602 0.298 0.398]
```

Internal Validation

Internal validation means validating a clustering model by defining metrics that capture the expected behavior of good clustering model. A good clustering can be identify as :

- Compact groups, i.e. the data points in one cluster occur close to each other
- Well separated group, i.e. two groups/clusters have as large distance among them as possible

Silhouette coefficient is a metric that tries to combine the two requirement of good clustering model. The silhouette coefficient is defined for each sample and is a

combination of its similarity to the data points in its own cluster and its dissimilarity to the data points not in its cluster.

24. In the scikit-learn, we can compute the Silhouette coefficient using the silhouette_score function. The function also allows for different options for distance metrics.

```
from sklearn import metrics

km2_silc = metrics.silhouette_score(X, km2_labels, metric='euclidean')
km5_silc = metrics.silhouette_score(X, km5_labels, metric='euclidean')

print('Silhouette Coefficient for num clusters=2: ', km2_silc)
print('Silhouette Coefficient for num clusters=5: ', km5_silc)
```

```
Silhouette Coefficient for num clusters=2:  0.6972646156059464
Silhouette Coefficient for num clusters=5:  0.5102292997907839
```

Evaluating Regression Model

Regression model are an example of supervised learning methods and owning to the availability of the correct measures (real valued numeric response variables), their evaluations is relatively easier than unsupervised models. Usually in the case of supervised models, we are spoilt for the choice of metrics and the important decision is choosing the right one for our use case. In this evaluating, we use Mean Square Error and R squared (R^2).

Mean Square Error (MSE)

MSE calculate the average of the squares of the errors or deviation between the actual value and the predicted value, as predicted by a regression model. The MSE can be used to evaluated a regression model, with lower values meaning a better regression models with less error. The following is the example of using of MSE.

25. Load the libraries

```
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
```

26. Generate feature matrix, generate target vector

```
feature, target = make_regression(n_samples=100,
                                  n_features=3,
                                  n_informative=3,
                                  n_targets =1,
                                  noise=1,
                                  coef=False,
                                  random_state=1)
```

27. Create a linear regression object

```
ols = LinearRegression()
```

28. Cross validate the linear regression using (negative) MSE

```
cross_val_score(ols,feature, target, scoring='neg_mean_squared_error')
```

```
array([-0.68729127, -1.24136497, -0.55087144])
```

R-Squared (R^2)

R-Squared or coefficient of determination measures the proportion of variance in the dependent variable which is explained by the independent variable. A coefficient of determination score of 1 denotes a perfect regression model indicating that all of the variance is explained by the independent variables.

29. The function of R-squared in scikit-learn is using scoring = ‘r2’. This the following code :

```
cross_val_score(ols,feature,target,scoring='r2')
```

```
array([0.99994596, 0.99985948, 0.99994802])
```

MODEL TUNING

Model tuning is one of the most important concepts of machine learning and it does require some knowledge of the underlying math and logic of the algorithm in focus. We start by introducing these so-called parameters that are associated with machine learning algorithms, then we try to justify why it is hard to have a perfect model.

So, the aim is we try to optimize the hyperparameters. What are hyperparameters? The simplest definition is that hyperparameters are meta parameters that are associated with any machine learning algorithm and are usually set before the model training and building process. We do this because hyperparameters do not have any dependency on being derived from the underlying dataset on which a model is trained. Hyperparameters are extremely important for tuning the performance of learning algorithms.

Hyperparameter Tuning Strategies

The simplest of the hyperparameter optimization methods is Grid Search method. In this method we will specify the grid of values (of hyperparamters) that we want to try out and optimize to get the best parameter combinations.

30. Let's first split our breast cancer dataset variables X and y into train and test datasets and build an SVM model with default parameters. Then we'll evaluate its performance on the test dataset by leveraging our *model_evaluation_utils* module.

```
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

# prepare datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# build default SVM model
def_svc = SVC(random_state=42)
def_svc.fit(X_train, y_train)

# predict and evaluate performance
def_y_pred = def_svc.predict(X_test)
print('Default Model Stats:')
meu.display_model_performance_metrics(true_labels=y_test, predicted_labels=def_y_pred, classes=[0,1])
```

```
Default Model Stats:
Model Performance metrics:
------------------------------
Accuracy: 0.6316
Precision: 0.3989
Recall: 0.6316
F1 Score: 0.489

Model Classification report:
------------------------------
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        63
           1       0.63      1.00      0.77       108

   micro avg       0.63      0.63      0.63       171
   macro avg       0.32      0.50      0.39       171
weighted avg       0.40      0.63      0.49       171


Prediction Confusion Matrix:
------------------------------
           Predicted:
                     0    1
Actual: 0            0   63
        1            0  108
```

31. Setting the parameter grid, which is C (deals with the margin parameter in SVM), the kernel function (used for transforming data into a higher dimensional feature space), and gamma (determines the influence a single training data point has).

```
from sklearn.model_selection import GridSearchCV

# setting the parameter grid
grid_parameters = {'kernel': ['linear', 'rbf'],
                   'gamma': [1e-3, 1e-4],
                   'C': [1, 10, 50, 100]}

# perform hyperparameter tuning
print("# Tuning hyper-parameters for accuracy\n")
clf = GridSearchCV(SVC(random_state=42), grid_parameters, cv=5, scoring='accuracy')
clf.fit(X_train, y_train)
# view accuracy scores for all the models
print("Grid scores for all the models based on CV:\n")
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
    print("%0.5f (+/-%0.05f) for %r" % (mean, std * 2, params))
# check out best model performance
print("\nBest parameters set found on development set:", clf.best_params_)
print("Best model validation accuracy:", clf.best_score_)
```

```
# Tuning hyper-parameters for accuracy

Grid scores for all the models based on CV:

0.95226 (+/-0.06310) for {'C': 1, 'gamma': 0.001, 'kernel': 'linear'}
0.91206 (+/-0.04540) for {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'}
0.95226 (+/-0.06310) for {'C': 1, 'gamma': 0.0001, 'kernel': 'linear'}
0.92462 (+/-0.02338) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.96231 (+/-0.04297) for {'C': 10, 'gamma': 0.001, 'kernel': 'linear'}
0.90201 (+/-0.04734) for {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
0.96231 (+/-0.04297) for {'C': 10, 'gamma': 0.0001, 'kernel': 'linear'}
0.92965 (+/-0.03425) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
0.95729 (+/-0.05989) for {'C': 50, 'gamma': 0.001, 'kernel': 'linear'}
0.90201 (+/-0.04734) for {'C': 50, 'gamma': 0.001, 'kernel': 'rbf'}
0.95729 (+/-0.05989) for {'C': 50, 'gamma': 0.0001, 'kernel': 'linear'}
0.93467 (+/-0.02975) for {'C': 50, 'gamma': 0.0001, 'kernel': 'rbf'}
0.95477 (+/-0.05772) for {'C': 100, 'gamma': 0.001, 'kernel': 'linear'}
0.90201 (+/-0.04734) for {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}
0.95477 (+/-0.05772) for {'C': 100, 'gamma': 0.0001, 'kernel': 'linear'}
0.93216 (+/-0.04674) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}

Best parameters set found on development set: {'C': 10, 'gamma': 0.001, 'kernel': 'linear'}
Best model validation accuracy: 0.9623115577889447
```

32. Thus, from the preceding output and code, we can see how the best model parameters were obtained based on cross validation accuracy and we get pretty awesome validation accuracy of 96%. Let's take this optimized and tuned model and put it to the test on our test data.

```
gs_best = clf.best_estimator_
tuned_y_pred = gs_best.predict(X_test)

print('\n\nTuned Model Stats:')
meu.display_model_performance_metrics(true_labels=y_test,
                                      predicted_labels=tuned_y_pred, classes=[0,1])
```

```
Tuned Model Stats:
Model Performance metrics:
------------------------------
Accuracy: 0.9708
Precision: 0.9709
Recall: 0.9708
F1 Score: 0.9708

Model Classification report:
------------------------------
              precision    recall  f1-score   support

           0       0.95      0.97      0.96        63
           1       0.98      0.97      0.98       108

   micro avg       0.97      0.97      0.97       171
   macro avg       0.97      0.97      0.97       171
weighted avg       0.97      0.97      0.97       171


Prediction Confusion Matrix:
------------------------------
          Predicted:
                   0    1
Actual: 0         61    2
        1          3  105
```

Well, things are certainly looking great now. Our model gives an overall F1 score and model accuracy of 97% on the test dataset too.

33. The another strategy is Randomized Search. Randomized parameter search is a modification to the traditional grid search. It takes input for grid elements as in normal grid search but it can also take distributions as input. To illustrate the use of randomized parameter search, we will use the example we used earlier but replace the gamma and C values with a distribution.

```
import scipy
from sklearn.model_selection import RandomizedSearchCV

param_grid = {'C': scipy.stats.expon(scale=10),
              'gamma': scipy.stats.expon(scale=.1),
              'kernel': ['rbf', 'linear']}

random_search = RandomizedSearchCV(SVC(random_state=42), param_distributions=param_grid,
                                   n_iter=50, cv=5)
random_search.fit(X_train, y_train)

print("Best parameters set found on development set:")
random_search.best_params_
```

```
Best parameters set found on development set:

{'C': 14.573819593159609, 'gamma': 0.09245672136619626, 'kernel': 'linear'}
```

```
rs_best = random_search.best_estimator_
rs_y_pred = rs_best.predict(X_test)
meu.get_metrics(true_labels=y_test, predicted_labels=rs_y_pred)
```

```
Accuracy: 0.9591
Precision: 0.9593
Recall: 0.9591
F1 Score: 0.9591
```

MODEL INTERPRETATION

We will use another library named ***Skater***, an open source Python library design to demystify the inner workings of predictive models. Skater defines the scope of interpretating models 1) globally (on the basis of a complete dataset) and 2) locally (on the basis of an individual prediction)

34. Installing skater

```
conda install -c conda-forge skater
```

```
Collecting package metadata (repodata.json): ...working... done
Solving environment: ...working... done

# All requested packages already installed.
```

35. We will be using our train and test dataset from the breast cancer dataset. We will leverage the ***X_train*** and ***X_test*** variables and also the logistic model object (logistic regression model) that we created previously. We will try to run some model interpretations on this model objects.

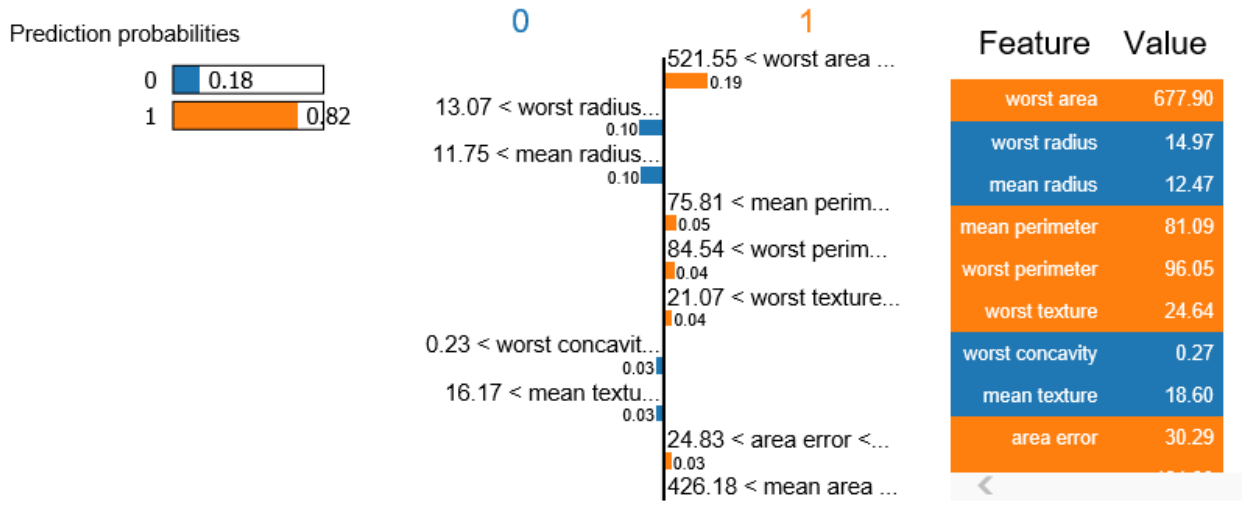
```
#Model interpretation
from skater.core.explanations import Interpretation
from skater.model import InMemoryModel

interpreter = Interpretation(X_test, feature_names=data.feature_names)
model = InMemoryModel(logistic.predict_proba, examples=X_train,
                      target_names=logistic.classes_)
```

36. Let's try interpret some predictions. We will predict two data points, one not having cancer (label 1) and one having cancer (label 0) and try to interpret the prediction making process.

```
#Explaining Prediction
from skater.core.local_interpretation.lime.lime_tabular import LimeTabularExplainer
exp = LimeTabularExplainer(X_train, feature_names=data.feature_names,
                           discretize_continuous=True, class_names=['0', '1'])
```

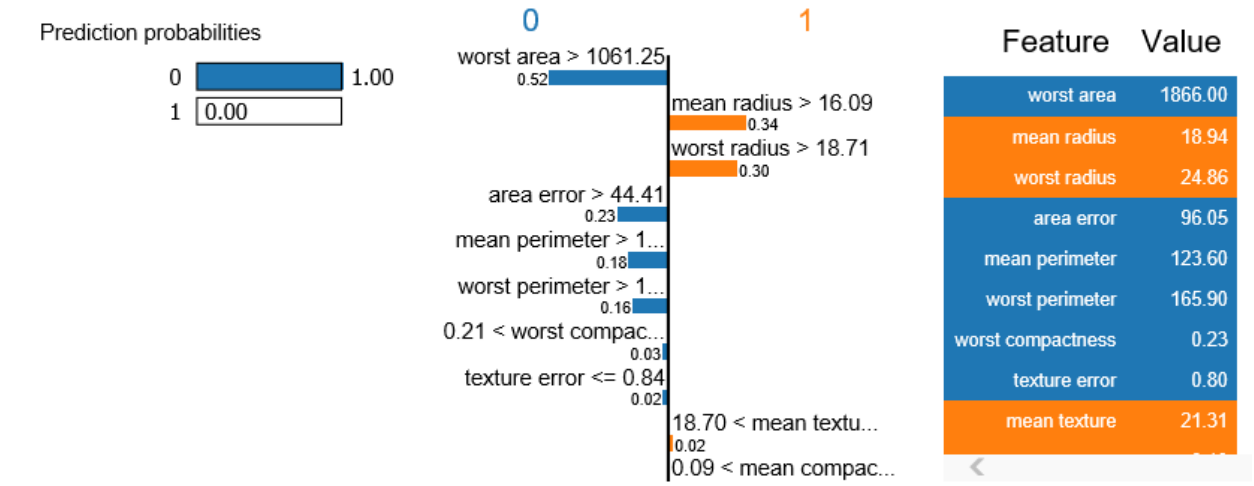
37. Explain prediction for data point having no cancer, i.e label 1

```
exp.explain_instance(X_test[0], logistic.predict_proba).show_in_notebook()
```



38. Explain prediction for data point having malignant cancer, i.e label 0

```
1 exp.explain_instance(X_test[1], logistic.predict_proba).show_in_notebook()
```



MODEL DEPLOYMENT

The tough part pf the whole modeling process is mostly the iterative process of feature engineering, model building, tuning and evaluation. Once we are done with this iterative process of model development, we can breathe a sigh of relief – but not for long. The final piece of Machine Learning modelling puzzle is that of deploying the model in production so that we actually start using it.

39. Model persistence is the simplest way of deploying a model. In this scheme of things we will persist out our final model on permanent media like our hard drive and use this persisted version for making predictions in the future. For persisting our model to disk, we can leverage libraries like ***pickle*** or ***joblib***, which is also available with scikit-learn. This allows us to deploy and use the model in the future, without having to retrain it each time we want to use it.

```
1 #Model Deployment
2
3 from sklearn.externals import joblib
4 joblib.dump(logistic, 'lr_model.pkl')
```

```
['lr_model.pkl']
```

This code will persist our model on the disk as a file named **lr_model.pkl.**

40. If we load this object in memory again we will get the logistic regression model object.

```
lr=joblib.load('lr_model.pkl')
lr
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)
```

41. We can now use lr object, which is our model loaded from the disk, and make prediction. A sample is depicted as follows.

```
print(lr.predict(X_test[10:11]), y_test[10:11])
```

```
[1] [1]
```

Referensi :

Raschka S, Mirjailili. 2017. Phython Machine Leraning 2nd edition. Packt Publishing