

# P. VAISHNAVI

## SVM (Support Vector Machines)

you will use SVM (Support Vector Machines) to build and train a model using human cell records, and classify cells to whether the samples are benign or malignant.

SVM works by mapping data to a high-dimensional feature space so that data points can be categorized, even when the data are not otherwise linearly separable. A separator between the categories is found, then the data is transformed in such a way that the separator could be drawn as a hyperplane. Following this, characteristics of new data can be used to predict the group to which a new record should belong.

```
In [34]: import pandas as pd
import pylab as pl
import numpy as np
import scipy.optimize as opt
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
%matplotlib inline
import matplotlib.pyplot as plt
```

- Field name Description
- ID Clump thickness
- Clump Clump thickness
- UnifSize Uniformity of cell size
- UnifShape Uniformity of cell shape
- MargAdh Marginal adhesion
- SingEpiSize Single epithelial cell size
- BareNuc Bare nuclei
- BlandChrom Bland chromatin
- NormNucl Normal nucleoli
- Mit Mitoses
- Class Benign or malignant

## Load Data From CSV File

```
In [35]: cell_df = pd.read_csv("cell_samples.csv")
cell_df.head()
```

```
Out[35]:
```

	ID	Clump	UnifSize	UnifShape	MargAdh	SingEpiSize	BareNuc	BlandChrom	NormNucl	Mi
0	1000025	5	1	1	1	2	1	3	1	
1	1002945	5	4	4	5	7	10	3	2	
2	1015425	3	1	1	1	2	2	3	1	

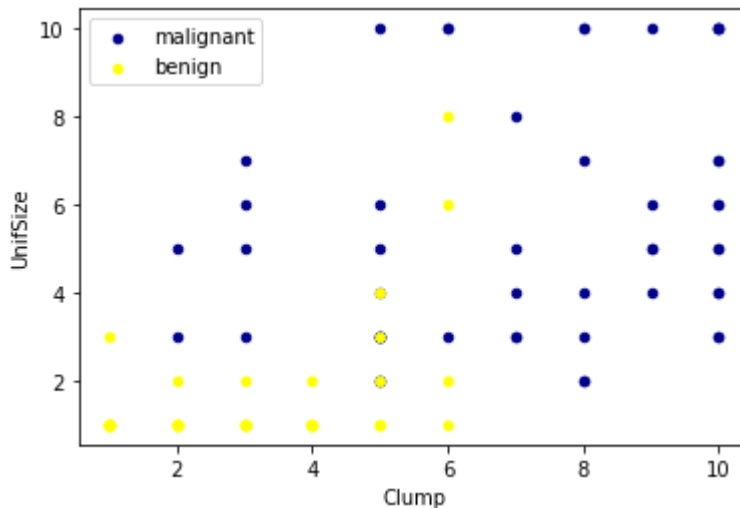
	ID	Clump	UnifSize	UnifShape	MargAdh	SingEpiSize	BareNuc	BlandChrom	NormNucl	Mi
3	1016277	6	8	8	1	3	4	3	7	
4	1017023	4	1	1	3	2	1	3	1	

The ID field contains the patient identifiers. The characteristics of the cell samples from each patient are contained in fields Clump to Mit. The values are graded from 1 to 10, with 1 being the closest to benign.

The Class field contains the diagnosis, as confirmed by separate medical procedures, as to whether the samples are benign (value = 2) or malignant (value = 4).

Lets look at the distribution of the classes based on Clump thickness and Uniformity of cell size:

```
In [36]: ax = cell_df[cell_df['Class'] == 4][0:50].plot(kind='scatter', x='Clump', y='UnifSize',
cell_df[cell_df['Class'] == 2][0:50].plot(kind='scatter', x='Clump', y='UnifSize', color='yellow')
plt.show()
```



## Data pre-processing and selection

Lets first look at columns data types:

```
In [37]: cell_df.dtypes
```

```
Out[37]: ID                int64
Clump                int64
UnifSize            int64
UnifShape            int64
MargAdh              int64
SingEpiSize          int64
BareNuc              object
BlandChrom            int64
NormNucl              int64
Mit                  int64
Class                int64
dtype: object
```

It looks like the **BareNuc** column includes some values that are not numerical. We can drop those rows:

```
In [38]: cell_df = cell_df[pd.to_numeric(cell_df['BareNuc'], errors='coerce').notnull()]
cell_df['BareNuc'] = cell_df['BareNuc'].astype('int')
cell_df.dtypes
```

```
Out[38]: ID          int64
Clump          int64
UnifSize       int64
UnifShape      int64
MargAdh        int64
SingEpiSize    int64
BareNuc        int64
BlandChrom     int64
NormNuc1       int64
Mit            int64
Class          int64
dtype: object
```

```
In [39]: X = cell_df.iloc[:, 2: 9].values
y = cell_df.iloc[:, 10].values
```

```
In [40]: print (X)
```

```
[[ 1  1  1 ...  1  3  1]
 [ 4  4  5 ... 10  3  2]
 [ 1  1  1 ...  2  3  1]
 ...
 [10 10  3 ...  3  8 10]
 [ 8  6  4 ...  4 10  6]
 [ 8  8  5 ...  5 10  4]]
```

```
In [42]: print (y)
```

```
[2 2 2 2 2 4 2 2 2 2 2 2 4 2 4 4 2 2 4 2 4 4 2 2 4 2 2 2 2 2 4 2 2 2 4 2
 4 4 4 4 4 4 2 4 2 2 4 4 4 4 4 4 4 4 4 4 2 4 4 2 4 2 4 4 2 2 4 2 4 4 2
 2 2 2 2 2 2 2 2 4 4 4 4 2 2 2 2 2 2 2 2 2 4 4 4 4 2 4 4 4 4 2 4 2 4 4
 4 2 2 2 4 2 2 2 2 4 4 4 2 4 2 4 2 2 2 4 2 2 2 2 2 2 2 2 4 2 2 4 2 2 4 2 4
 4 2 2 4 2 2 4 4 2 2 2 2 4 4 2 2 2 2 4 4 4 2 4 2 4 2 2 2 4 4 2 4 4 4 2 4
 4 2 2 2 2 2 2 2 2 4 4 2 2 2 4 4 2 2 2 4 4 2 4 4 4 2 2 4 2 2 4 4 4 2 4 4
 2 4 4 4 2 4 2 4 4 4 4 2 2 2 2 2 2 4 4 2 2 4 2 4 4 4 2 2 2 2 4 4 4 4 2 4
 4 4 2 4 2 4 4 2 2 2 2 4 2 2 4 4 4 4 4 2 2 4 4 2 2 4 4 2 2 4 4 2 4 4 2 2
 4 2 2 2 4 2 2 4 2 2 4 2 2 4 2 2 4 4 4 2 2 4 4 2 2 4 4 2 2 4 4 2 2 2 2
 2 4 4 2 2 2 4 2 2 4 4 4 4 4 2 2 2 2 4 4 2 2 2 2 2 2 2 2 2 2 2 2 4 2 2
 2 2 4 2 2 2 2 4 2 2 2 2 2 2 2 2 4 2 2 2 2 2 2 2 2 2 2 2 4 2 4 2 2 2 4
 2 2 2 4 2 4 2 2 2 2 2 2 2 4 4 2 2 2 4 2 2 2 2 2 2 2 2 2 4 2 2 4 4 2
 2 2 2 2 2 2 2 2 2 2 2 2 4 2 2 2 2 2 2 4 4 2 2 2 2 2 2 2 2 2 4 2 2 4 4
 4 4 2 2 4 2 2 2 2 2 2 4 4 2 2 2 4 2 4 4 4 2 4 2 2 2 2 2 2 2 2 4 4 4 2
 2 4 2 4 4 4 2 2 2 2 2 2 2 2 2 2 4 2 2 2 2 2 2 2 4 2 2 4 2 2 2 2 2 2 2
 2 2 2 4 2 2 2 2 2 2 2 2 4 2 2 2 2 2 2 2 2 2 2 4 4 4 2 2 2 2 2 2 2 4 4
 2 2 2 2 2 2 2 2 2 2 4 2 2 2 2 4 4 4]
```

We want the model to predict the value of Class (that is, benign (=2) or malignant (=4)). As this field can have one of only two possible values, we need to change its measurement level to reflect this.

# Train/Test dataset

Okay, we split our dataset into train and test set:

```
In [43]: # Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state
```

```
In [44]: cell_df.shape
```

```
Out[44]: (683, 11)
```

```
In [45]: print("The shape of training cell_df is", X_train.shape)
print("The shape of test data example is", X_test.shape)
```

The shape of training cell\_df is (546, 7)  
The shape of test data example is (137, 7)

## Modeling (SVM with Scikit-learn)

The SVM algorithm offers a choice of kernel functions for performing its processing. Basically, mapping data into a higher dimensional space is called kernelling. The mathematical function used for the transformation is known as the kernel function, and can be of different types, such as:

- 1.Linear
- 2.Polynomial
- 3.Radial basis function (RBF)
- 4.Sigmoid

Each of these functions has its characteristics, its pros and cons, and its equation, but as there's no easy way of knowing which function performs best with any given dataset, we usually choose different functions in turn and compare the results. Let's just use the default, RBF (Radial Basis Function) for this lab.

```
In [46]: from sklearn import svm
clf = svm.SVC(kernel='rbf')
clf.fit(X_train, y_train)
```

```
Out[46]: SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
max_iter=-1, probability=False, random_state=None, shrinking=True,
tol=0.001, verbose=False)
```

After being fitted, the model can then be used to predict new values:

```
In [47]: yhat = clf.predict(X_test)
yhat [0:5]
```

```
Out[47]: array([2, 2, 4, 4, 2])
```

# Evaluation

```
In [48]: from sklearn.metrics import classification_report, confusion_matrix
import itertools
```

```
In [49]: def plot_confusion_matrix(cm, classes,
                                normalize=False,
                                title='Confusion matrix',
                                cmap=plt.cm.Blues):

    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

```
In [51]: # Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, yhat, labels=[2,4])
np.set_printoptions(precision=2)

print (classification_report(y_test, yhat))

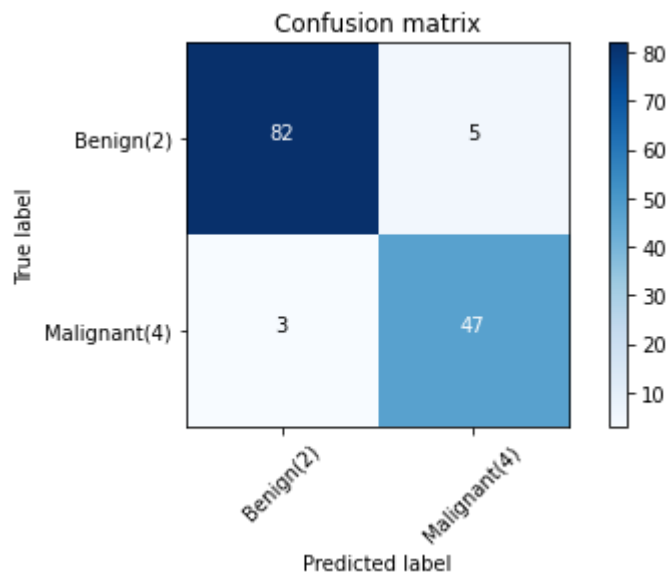
# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=['Benign(2)', 'Malignant(4)'], normalize= False
```

	precision	recall	f1-score	support
2	0.96	0.94	0.95	87
4	0.90	0.94	0.92	50
accuracy			0.94	137
macro avg	0.93	0.94	0.94	137

weighted avg      0.94      0.94      0.94      137

Confusion matrix, without normalization

```
[[82  5]
 [ 3 47]]
```



You can also easily use the **f1\_score** from sklearn library:

```
In [53]: from sklearn.metrics import f1_score
print("Avg F1-score: %.4f" % f1_score(y_test, yhat, average='weighted'))
```

Avg F1-score: 0.9418

Lets try jaccard index for accuracy:

```
In [55]: from sklearn.metrics import jaccard_similarity_score
print("Jaccard score: %.4f" % jaccard_similarity_score(y_test, yhat))
```

Jaccard score: 0.9416

/usr/local/lib/python3.7/dist-packages/sklearn/metrics/\_classification.py:664: FutureWarning: jaccard\_similarity\_score has been deprecated and replaced with jaccard\_score. It will be removed in version 0.23. This implementation has surprising behavior for binary and multiclass classification tasks.  
FutureWarning)