SVM (Support Vector Machines)

Objetivos

- Usar SVM para clasificar

Utilizaremos SVM para construir y entrenar un modelo usando registros de células humanas y clasificarlas como benignas o malignas.

SVM funciona mapeando los datos a un espacio de características de dimensión mayor para que los puntos de datos puedan ser categorizados, aún cuando los datos no sean linealmente separables. Se encuentra un separador entre las categorías y luego los datos son transformados de tal forma que el separador pueda dibujarse como un hiperplano. Finalmente, las características de los nuevos datos pueden utilizarse para predecir el grupo al cual pertenece un nuevo registro.

Tabla de contenido

In [1]:
```
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
```

Cargando los datos

El ejemplo se basa en un dataset público disponible desde el repositorio UCI Machine Learning [http://mlearn.ics.uci.edu/MLRepository.html]. El dataset consiste de varios cientos de muestras de células humanas, cada una de las cuales contiene los valores de un conjunto de características. Los campos en cada registro son:

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

Para este ejemplo, usaremos un dataset que contiene un número pequeño de predictores en cada registro.

In [2]:
```
#Click here and press Shift+Enter
#wget -O cell_samples.csv https://cf-courses-data.s3.us.cloud-object-storage.appdoma

import urllib.request
url = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDevelope
filename = 'cell_samples.csv'
urllib.request.urlretrieve(url, filename)
```

Out[2]:
```
('cell_samples.csv', <http.client.HTTPMessage at 0x1cb6467f160>)
```

Cargando datos desde el CSV

In [4]:
```
cell_df = pd.read_csv("cell_samples.csv")
cell_df.head()
```

Out[4]:

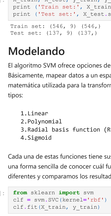
	ID	Clump	UnifSize	UnifShape	MargAdh	SingEpiSize	BareNuc	BlandChrom	NormNucl	Mit	Class
0	1000025	5	1	1	1	2	1	3	1	1	2
1	1002945	5	4	4	5	7	10	3	2	1	2
2	1015425	3	1	1	1	2	2	3	1	1	2
3	1016277	6	8	8	1	3	4	3	7	1	2
4	1017023	4	1	1	3	2	1	3	1	1	2

El ID contiene los identificadores de pacientes. Las características de las muestras de células de cada paciente están contenidas entre los campos *Clump* y *Mit*. Los valores van de 1 a 10, siendo 1 lo más cerca a benigno.

El campo Clase contiene el diagnóstico, confirmado por procedimientos médicos separados, en cuanto a si las muestras son benignas (valor 2) o malignas (valor no 4).

Veamos la distribución de las clases en función del grosor del grupo y la uniformidad del tamaño de celda:

In [5]:
```
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', co
plt.show()
```



Pre-procesamiento de datos y selección

Miremos los tipos de datos de las columnas:

In [6]:
```
cell_df.dtypes
```

Out[6]:
```
ID             int64
Clump          int64
UnifSize       int64
UnifShape      int64
MargAdh        int64
SingEpiSize    int64
BareNuc       object
BlandChrom     int64
NormNucl       int64
Mit            int64
Class          int64
dtype: object
```

Parece que **BareNuc** incluye algunos valores que no son numéricos. Descartamos estas columnas:

In [7]:
```
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[7]:
```
ID             int64
Clump          int64
UnifSize       int64
UnifShape      int64
MargAdh        int64
SingEpiSize    int64
BareNuc        int32
BlandChrom     int64
NormNucl       int64
Mit            int64
Class          int64
dtype: object
```

In [8]:
```
feature_df = cell_df[['Clump', 'UnifSize', 'UnifShape', 'MargAdh', 'SingEpiSize', 'Ba
X = np.asarray(feature_df)
X[0:5]
```

Out[8]:
```
array([[ 5.,  1.,  1.,  1.,  2.,  1.,  3.,  1.,  1.],
       [ 5.,  4.,  4.,  5.,  7., 10.,  3.,  2.,  1.],
       [ 3.,  1.,  1.,  1.,  2.,  2.,  3.,  1.,  1.],
       [ 6.,  8.,  8.,  1.,  3.,  4.,  3.,  7.,  1.],
       [ 4.,  1.,  1.,  3.,  2.,  1.,  3.,  1.,  1.]], dtype=int64)
```

Queremos que el modelo prediga el valor de Clase (es decir, benigno (2) o maligno (4)). Como este campo puede tener uno de los dos únicos valores posibles, necesitamos cambiar su nivel de medición para reflejar esto.

In [9]:
```
cell_df['Class'] = cell_df['Class'].astype('int')
y = np.asarray(cell_df['Class'])
y [0:5]
```

Out[9]:
```
array([2, 2, 2, 2, 2])
```

Train/Test dataset

Dividimos en conjuntos de entrenamiento y test:

In [10]:
```
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_stat
print ('Train set:', y_train.shape,  y_train.shape)
print ('Test set:', X_test.shape,  y_test.shape)
```

```
Train set: (546, 9) (546,)
Test set: (137, 9) (137,)
```

Modelando

El algoritmo SVM ofrece opciones de funciones de núcleo (kernel) para realizar su procesamiento. Básicamente, mapear datos a un espacio de mayor dimensionalidad se llama kernelling. La función matemática utilizada para la transformación se conoce como función de kernel, y puede tener distintos tipos:

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

Cada una de estas funciones tiene sus características, sus pros y contras y su ecuación, pero como no hay una forma sencilla de conocer cuál funcionará mejor para un dataset dado usualmente elegimos funciones diferentes y comparamos los resultados. Usemos la función por defecto (RBF) para este laboratorio.

In [11]:
```
from sklearn import svm
clf = svm.SVC(kernel='rbf')
clf.fit(X_train, y_train)
```

Out[11]:
```
SVC()
```

Luego de ajustar el modelo, podemos utilizarlo para predecir nuevos valores:

In [12]:
```
yhat = clf.predict(X_test)
yhat [0:5]
```

Out[12]:
```
array([2, 4, 2, 4, 2])
```

Evaluación

In [13]:
```
from sklearn.metrics import classification_report, confusion_matrix
import itertools
```

In [14]:
```
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 [15]:
```
# 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= Fal
```

```
              precision    recall  f1-score   support

           2       1.00      0.94      0.97        90
           4       0.90      1.00      0.95        47

    accuracy                           0.96       137
   macro avg       0.95      0.97      0.96       137
weighted avg       0.97      0.96      0.96       137

Confusion matrix, without normalization
[[85  5]
 [ 0 47]]
```

Podemos usar **f1_score** de sklearn:

In [16]:
```
from sklearn.metrics import f1_score
f1_score(y_test, yhat, average='weighted')
```

Out[16]:
```
0.9639038982104676
```

Intentemos con Jaccard para la precisión:

In [20]:
```
from sklearn.metrics import jaccard_score
#jaccard_score(y_test, yhat)
```

Práctica

Reconstruyamos el modelo utilizando una función de kernel lineal. Veamos cómo cambia la precisión.

In [22]:
```
clf2 = svm.SVC(kernel='linear')
clf2.fit(X_train, y_train)
yhat2 = clf2.predict(X_test)
print("Avg F1-score: %.4f" % f1_score(y_test, yhat2, average='weighted'))
#print("Jaccard score: %.4f" % jaccard_score(y_test, yhat2))
```

```
Avg F1-score: 0.9639
```