Minería de datos: máquinas de vectores soporte

Andrés Mañas Mañas

1 de junio de 2017

Índice

1. Resumen teórico

Las máquinas de soporte vectorial son un tipo de algoritmos utilizados principalmente para clasificación y para regresión desarollados por Vladimir Vapnik y su equipo [1].

Las máquinas de soporte vectorial funcionan generando un hiperplano que separa el espacio de muestras de forma que, idealmente, patrones de diferentes clases nunca aparezcan en el mismo lado del plano. Como existen infinitos hiperplanos que separen el espacio, se busca aquel cuya distancia entre el hiperplano y los patrones de cada clase sea la máxima posible (ver Figura 1). A los puntos que conforman las dos líneas paralelas al hiperplano, siendo la distancia entre ellas (margen) la mayor posible, se les denomina vectores de soporte. En la fase de entrenamiento de las máquinas de soporte vectorial, nos centraremos en hallar los vectores de soporte empleando técnicas de programación cuadrática [3].

Una vez obtenemos el hiperplano podemos utilizar la Máquina en problemas de clasificación, escogiendo patrones que no hayan sido vistos durante la fase de entrenamiento, y aplicando una etiqueta (clase) en función del lado del hiperplano en el que se proyecten.

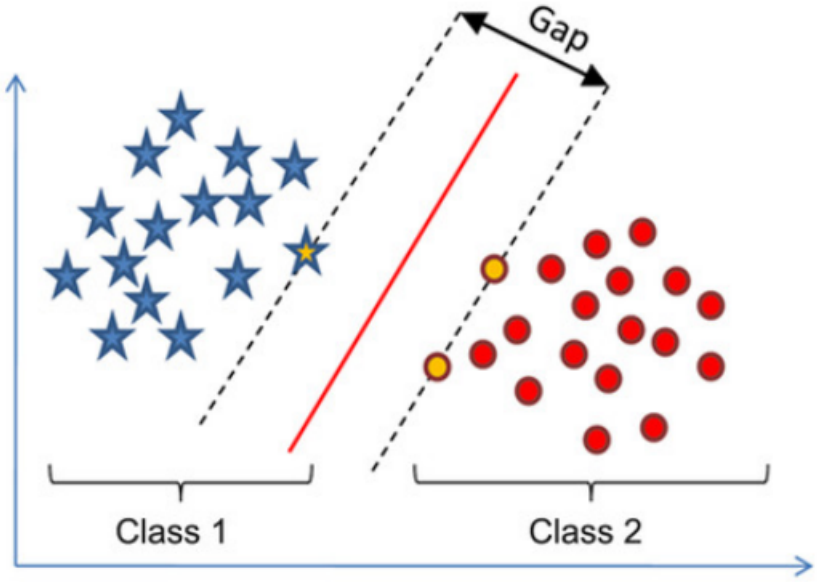


Figura 1: Ejemplo de 2 dimensiones para una máquina de soporte vectorial, obsérvese que la recta roja es la que maximiza la distancia entre los vectores de soporte (los puntos en amarillo)

Los universos a estudiar no se suelen presentar en casos idílicos de dos dimensiones como en el ejemplo anterior, sino que un algoritmo SVM debe tratar con:

- Más de dos variables predictoras
- Curvas no lineales de separación
- Casos donde los conjuntos de datos no pueden ser completamente separados
- Clasificaciones en más de dos categorías

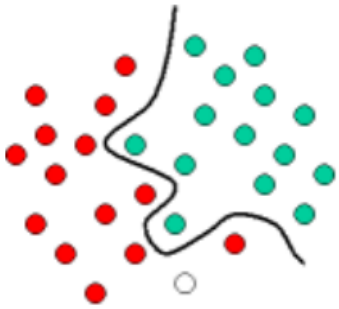


Figura 2: Espacio de 2 dimensiones en el que los patrones no son separables linealmente

1.1. Definición formal

Dado un conjunto de entrenamiento formado por pares de instancias y etiquetas

$$(x_i, y_i), i = 1, ..., l \text{ donde } x_i \in R^n, y \in \{1, -1\}^l$$

, las SVMs pueden entenderse como la solución al siguiente problema de optimización::

$$\begin{aligned} \underset{w,b,\epsilon}{\text{argmin}} \quad & \frac{1}{2} w^T w + C \sum_1^l \epsilon_i \\ \text{sujeto a} \quad & y_i(w^T \phi(x_i) + b) \geq 1 - \epsilon_i \\ & \epsilon_i \geq 0 \end{aligned}$$

Además,

$$K(x_i, x_j) \equiv \phi(x_i)^T \phi(x_j)$$

es llamada la función kernel.

Figura 3. Definición formal de las SVMs según [4]

Los vectores de entrenamiento x_i son asignados a un espacio de mayor dimensiones (incluso infinitas) por la función ϕ. La máquina de soporte vectorial encuentra el hiperplano con máximo margen de separación en el espacio dimensional superior. C > 0 es el parámetro de penalización del término de error.

1.2. Funciones kernel

Cuando el espacio donde trabajamos no permite separar linealmente los patrones de forma perfecta, como se puede observar en la figura 2, es imposible trazar un hiperplano que separe perfectamente los patrones.

Por ello entran en juego las llamadas **funciones kernel**, funciones matemáticas que nos permiten proyectar los patrones en un espacio de mayor dimensión que el original dónde estos si son linealmente separables mediante un hiperplano (Figura 3).

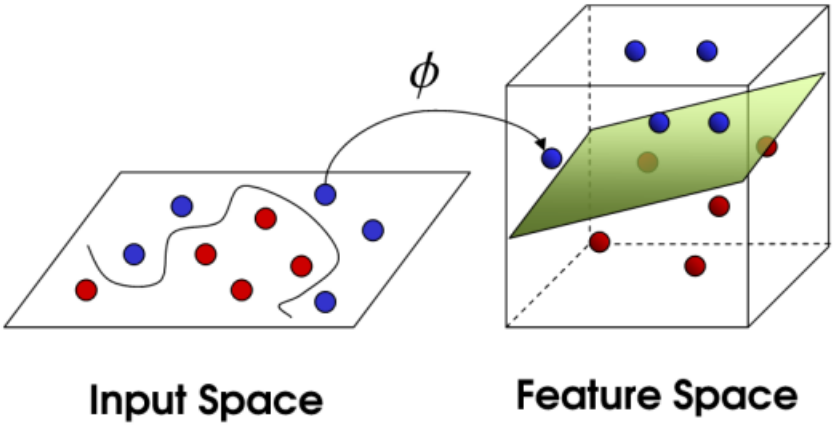


Figura 4: Kernel de 2 a 3 dimensiones dónde los patrones pueden ser perfectamente separados mediante un hiperplano

De nuevo, y como en la gran mayoría de algoritmos de aprendizaje máquina, **si elegimos un modelo (en este caso una función kernel) que se ajuste demasiado bien a los patrones de entrenamiento**, el clasificador perderá capacidad de generalización por lo que tendrá un mal comportamiento ante nuevos patrones y se hablará de **sobreentrenamiento**.

Las funciones kernel más comunmente utilizadas son:

- **lineales**: $K(x_i, x_j) = x_i T x_j$
- **polinómicas**: $K(x_i, x_j) = (\gamma x_i T x_j + r)d, \gamma \geq 0$
- **funciones de base radial (RBF)**: $K(x_i, x_j) = exp(-\gamma |x_i - x_j|^2), \gamma \geq 0$
- **sigmoide**: $K(x_i, x_j) = \tanh(\gamma x_i^T x_j + r)$

1.3. Outliers

No siempre es posible encontrar una transformación de los datos que permita separarlos linealmente [2] o incluso a veces, no es conveniente.

En este caso nos encontramos con instancias en el conjunto de entrenamiento que se situan en lugares del espacio muy separados o aislados de la zona dónde se sitúan la mayoría de instancias de entrenamiento con las que comparten la clase.

La presencia de ruido o de **outliers** pueden llegar tambien a provocar soluciones sobreajustadas a los patrones de entrenamiento que no generalizan bien. Para tratar con estas situaciones se crearon los llamados **SVM de margen blando** cuya idea principal es introducir una holgura al margen existente entre los vectores de soporte y el hiperplano que permite que las restricciones no se cumplan de manera estricta.

Se introduce por tanto en el proceso de entrenamiento una constante **C** que controlará el tamaño de la holgura.

El parámetro C le indica a la SVM, en el proceso de entrenamiento, el grado de ejemplos

mal clasificados permitido.

Para valores grandes de C, la optimización elegirá un hiperplano de margen más pequeño si ese hiperplano hace un mejor trabajo para conseguir que todos los puntos de entrenamiento estén clasificados correctamente. Es decir, se evitan en mayor medida ejemplos mal clasificados aunque ello conduzca a hiperplanos de margen menor.

Por el contrario, un valor muy pequeño de C hará que el optimizador busque un hiperplano de separación de mayor margen, incluso si ese hiperplano clasifica erróneamente más puntos. Para valores muy pequeños de C frecuentemente se obtienen ejemplos mal clasificados, incluso si los datos de entrenamiento son linealmente separables. Sin embargo, el sobreajuste del clasificador es menor y generalmente se obtienen clasificadores que generalizan mejor.

1.4. Preprocesado de datos

Se suele realizar a dos niveles [4]:

1.4.1. Atributos categóricos

Las SVMs requieren que cada instancia de datos se represente como un vector de números reales. Por lo tanto, si hay atributos categóricos, primero tenemos que convertirlos en datos numéricos. Se recomienda utilizar m números para representar un atributo con m categorías. Solamente uno de los m números es uno, y los otros son cero. Por ejemplo, una categoría de tres atributos como {rojo, verde, azul} puede representarse como (0,0,1), (0,1,0) y (1,0,0).

La experiencia indica que si el número de valores en un atributo no es demasiado grande, esta codificación puede ser más estable que usar un solo número.

1.4.2. Escalado

Escalar los datos antes de aplicar las SVMs es muy importante. La mayoría de las consideraciones hechas para redes neuronales también se aplican a las SVMs.

La principal ventaja de escalar es evitar que atributos cuyos valores se mueven en rangos numéricos altos dominen a aquellos que se mueven en rangos numéricos más pequeños. Otra ventaja es evitar dificultades numéricas durante el cálculo, dado que los valores del kernel normalmente dependen de los productos internos de vectores de características, p.

Se recomienda un escalado lineal de cada atributo al rango [-1, +1] o [0, 1]. Por supuesto, hay usar el mismo método para escalar el los datos de entrenamiento y los de prueba.

1.5. Selección del modelo

Aunque sólo son cuatro las funciones kernel más comunes, se debe decidir una para entrenar la SVM. Una vez elegido el núcleo, se ajustan los parámetros de penalización C y los parámetros propios del kernel elegido.

Según [4], en general, el núcleo RBF es una primera opción razonable.

Este kernel no correlaciona las muestras en un espacio dimensional superior, a diferencia del núcleo lineal, con lo que se puede manejar el caso cuando la relación entre las etiquetas y los atributos no es lineal. Además, el núcleo lineal es un caso especial de RBF.

La segunda razón es el número de hiperparámetros que influyen en la complejidad de la selección del modelo. El núcleo polinomial tiene más hiperparametros que el núcleo RBF.

Por último, el núcleo RBF tiene menos dificultades numéricas ya que el resultado de aplicar la función kernel es siempre menor que 1.

Hay algunas situaciones en las que el núcleo RBF no es adecuado. En particular, cuando el número de características es muy grande, se puede utilizar el núcleo lineal.

1.6. Cross-validation

Aprender los parámetros de un modelo y probarlo en los mismos datos es un error metodológico: un modelo que simplemente repitiera las etiquetas de las muestras que acaba de ver tendría una puntuación perfecta pero no podría predecir nada útil en otros datos no vistos. Esta situación se conoce como **sobreajuste**. Para evitarlo, es una práctica común cuando se realiza un experimento de aprendizaje (supervisado) mantener una parte de los datos disponibles como un conjunto de pruebas.

Una solución a este problema es un procedimiento llamado **validación cruzada** (CV para abreviar). Se guarda un conjunto de prueba para la evaluación final, que no se utiliza en ningún caso en el proceso de entrenamiento. En el enfoque básico, llamado **k-fold CV**, el conjunto de entrenamiento se divide en k conjuntos más pequeños. Se sigue el siguiente procedimiento para cada uno de los k "bloques", en diversas iteraciones:

- El modelo es entrenado usando k-1 de los bloques
- El modelo resultante se valida en el bloque restante no utilizado en esta iteración del entrenamiento
- Este proceso se repite k veces (una para utilizando cada bloque como prueba)

Típicamente, la medida de rendimiento reportada por la validación cruzada k-fold es el promedio de los valores calculados en las k iteraciones. Este enfoque puede ser costoso desde el punto de vista computacional, pero garantiza que los datos iniciales de prueba realmente se utilizan sólo para las pruebas y que el entrenamiento puede ser más productivo.

2. Notas a la implementación

En el enunciado de la práctica se hace mención explícita a WEKA y a Torch.

Sin embargo, entiendo que el lenguaje o la herramienta con la que resuelvan las prácticas es libre y las referencias a WEKA es simplemente por tener una herramienta por defecto para quien no tenga otras preferencias.

En este sentido, y espero no estar cometiendo un error, me tomo la libertad de desarrollar esta práctica con python utilizando scikit-learn como librería de machine-learning.

3. Análisis del fichero s

3.1. Cargar los ficheros s-train/test

```
from scipy.io.arff import loadarff

S_train,S_meta = loadarff('svm-data/s-train.arff')
S_test,_       = loadarff('svm-data/s-test.arff')
```

3.2. Función de normalización

Tal como se ha indicado en la parte teórica de la práctica, es fundamental normalizar los datos para obtener un correcto funcionamiento de las SVM's, y por extensión de casi cualquier algoritmo de minería de datos.

Aquí tenemos una función que nos hace todo el trabajo - **nótese que normalizamos simultáneamente los datos de entrenamiento y de prueba**. Sabemos que no aplicar el mismo tipo de normalización a los datos con los que se prueba un modelo que a aquellos con los que se entrena es tanto como probar el modelo con datos de otro dominio del que estudia.

```
from sklearn.preprocessing import normalize

def normalize_data(train, test):
    """Normaliza los datos de entrenamiento y prueba
    y los devuelve como una tupla:
    X_train,y_train,X_test,y_test
    """
    data = np.append(train, test)
    X = normalize([list(d)[0:-1] for d in data], axis=0)
    y = [list(d)[-1] for d in data]
    l = len(train)
    return X[:l],y[:l],X[l:],y[l:]
```

3.3. Seleccionar las 3 clases más frecuentes

```
import numpy as np


def top_three_categories(train,test):
    classes = np.concatenate((train['class'],test['class']))
    class_counts = dict(zip(*np.unique(classes,return_counts=True)))
    top_three = sorted(class_counts, key=class_counts.get, reverse=True)[0:3]
    print 'Top 3 classes: %s' % top_three
    return train[np.in1d(train['class'], top_three)],test[np.in1d(test['class'], top_three)]

S_top3_train,S_top3_test = top_three_categories(S_train,S_test)
```

```
Top 3 classes: ['46', '20', '110']
```

3.4. Obtención de un clasificador MVS lineal en el espacio de parámetros

Vamos a realizar una búsqueda en el espacio de parámetros para seleccionar el que produce mejores resultados evaluados mediante validación cruzada.

Según las indicacione de la práctica, probaremos unos 20 valores de **C entre 0.001 y 10**. Pero en lugar de utilizarlos equiespaciados en escala lineal, creo que será más productivo (ofrecerá mejor ámbito exploratorio) si los **equiespaciamos en escala logarítmica**.

```
from sklearn.model_selection import GridSearchCV
np.set_printoptions(formatter={'float': lambda x: "{0:0.3f}".format(x)})

N = 20
Cs = np.logspace(-3, 1, N)
print 'Cs: %s' % Cs
```

```
Cs: [0.001 0.002 0.003 0.004 0.007 0.011 0.018 0.030 0.048 0.078 0.127 0.207
 0.336 0.546 0.886 1.438 2.336 3.793 6.158 10.000]
```

Vamos a utilizar además una búsqueda exhaustiva en el conjunto de Cs anterior con la clase **GridSearchCV**.

Esta función realiza automáticamente el entrenamiento con cross-validation. Por defecto utiliza 3 folds, y por defecto lo dejo, ya que probando con otros no se obtienen resultados mucho mejores.

Después, obtendremos la información que necesitamos simplemente llamando a los miembros:

- **best_estimator_** : el modelo que ofreció los mejores resultados de todos los probados
- **best_score_** : la puntuación del mejor modelo indicado en el punto anterior

```
from sklearn import svm
from sklearn.model_selection import GridSearchCV

X_train,y_train,X_test,y_test = normalize_data(S_top3_train,S_top3_test)
gscv = GridSearchCV(estimator=svm.SVC(kernel='linear'),
                    param_grid=dict(C=Cs), n_jobs=-1)
gscv.fit(X_train, y_train)

print 'Mejor C: %s' % gscv.best_estimator_.C
print 'Puntuación del mejor modelo: %s' % gscv.best_score_
```

```
Mejor C: 10.0
Puntuación del mejor modelo: 0.835616438356
```

Como el mejor C enontrado se corresponde con el extremo derecho de los probados en nuestro **grid**, vamos a aumentar el intervalo de prueba por la derecha:

```
Cs = np.logspace(1, 3, N)
print "Cs: %s" % Cs
```

```
gscv = GridSearchCV(estimator=svm.SVC(kernel='linear'),
                    param_grid=dict(C=Cs), n_jobs=-1)
gscv.fit(X_train, y_train)
print 'Mejor C: %s' % gscv.best_estimator_.C
print 'Puntuación del mejor modelo: %s' % gscv.best_estimator_.score(X_test, y_test)
```

```
Cs: [10.000 12.743 16.238 20.691 26.367 33.598 42.813 54.556 69.519 88.587
 112.884 143.845 183.298 233.572 297.635 379.269 483.293 615.848 784.760
 1000.000]
Mejor C: 183.298071083
Puntuación del mejor modelo: 0.90756302521
```

Aunque el rendimiento de esta SVM es bastante bueno (90 %), resulta bastante incómodo que el valor C sea tan grande. Es decir, el mejor modelo encontrado **es de margen muy rígido**, lo que puede sugerir que el modelo está sobreajustado.

Para disipar esta duda, podemos hacer el mismo experimento pero probando con un valor más blando y ver si se obtienen resultados parecidos.

```
gscv = GridSearchCV(estimator=svm.SVC(kernel='linear'),
                    param_grid=dict(C=[0.1]), n_jobs=-1)
gscv.fit(X_train, y_train)
print 'Puntuación del modelo con C=0.1: %s' % gscv.best_estimator_.score(X_test, y_test)
```

```
Puntuación del modelo con C=0.1: 0.36974789916
```

Aunque hemos encontrado una máquina que ofrece unos resultados de clasificación en torno al 0.90 %, el parámetro de penalización de los errores de clasificación de la misma es altísimo, C=183. Sin embargo, probando coma un valor C=0.1 más pequeño, vemos que el rendimiento de la máquina obtenida es muy pequeño. Por lo tanto, entiendo que esto nos invita a pensar que el problema NO es linealmente separable. Sería linealmente separable si obtuviéramos buenos resultados de clasificación con una máquina de margen blando (C próximo a 0).

3.5. Realizar una búsqueda similar para núcleos polinómicos de grado 2

Procedemos de forma similar al punto anterior, pero en este caso utilizando una función kernel de tipo polinómica.

```
N = 20
Cs = np.logspace(-3, 1, N)
Gs = np.logspace(-3, 1, N)

for n in range(1,5):  # Probamos hasta grado 5
    gscv = GridSearchCV(estimator=svm.SVC(kernel='poly', degree=n),
                        param_grid=dict(C=Cs,gamma=Gs), n_jobs=-1)
    gscv.fit(X_train, y_train)
    print 'Con grado %s' % n
    print 'Mejor C: %s' % gscv.best_estimator_.C
    print 'Mejor gamma: %s' % gscv.best_estimator_.gamma
    print 'Puntuación del mejor modelo: %s' % gscv.best_estimator_.score(X_test, y_test)
```

```
Con grado 1
Mejor C: 10.0
Mejor gamma: 10.0
Puntuación del mejor modelo: 0.890756302521
Con grado 2
Mejor C: 10.0
Mejor gamma: 10.0
Puntuación del mejor modelo: 0.90756302521
Con grado 3
Mejor C: 3.79269019073
Mejor gamma: 10.0
Puntuación del mejor modelo: 0.915966386555
Con grado 4
Mejor C: 3.79269019073
Mejor gamma: 10.0
Puntuación del mejor modelo: 0.90756302521
```

Vemos que obtenemos una máquina capaz de clasificar el 91 % de las instancias con un C=3.79, es decir, de margen blando.

3.6. Obtención de un clasificador MVS lineal en el espacio proyectado mediante un núcleo gaussiano

Repetimos los pasos del apartado anterior incluyendo en la ventana de parámetros explorados dos entradas, una para el parámetro C y otra para G (gamma) del núcleo gaussiano o de base radial.

```
N = 20
Cs = np.logspace(-3, 1, N)
Gs = np.logspace(-3, 1, N)
gscv = GridSearchCV(estimator=svm.SVC(kernel='rbf'),
                    param_grid=dict(C=Cs, gamma=Gs), n_jobs=-1)
gscv.fit(X_train, y_train)
print 'Mejor C: %s' % gscv.best_estimator_.C
print 'Mejor gamma: %s' % gscv.best_estimator_.gamma
print 'Puntuación del mejor modelo: %s' % gscv.best_estimator_.score(X_test, y_test)
```

```
Mejor C: 3.79269019073
Mejor gamma: 3.79269019073
Puntuación del mejor modelo: 0.873949579832
```

En este caso observamos que aunque el scoring obtenido por el mejor modelo encontrado es similar al mejor encontrado en el caso anterior, el parámetro gamma es inferior. Entiendo que podemos pensar que para este conjunto de datos, SVMs basadas en núcleos gausianos son más apropiadas.

4. Probar distintos tipos de nucleos con un problema no separable linealmente

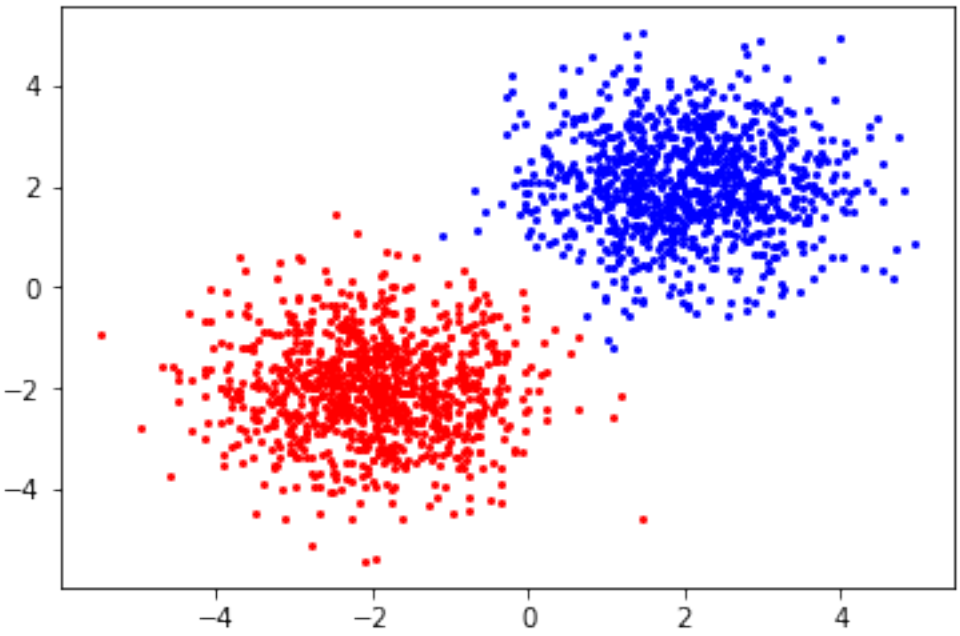
Definimos unas funciones que nos permita dibujar los datos y las curvas de decisión de los clasificadores obtenidos en cada experimento.

```
%matplotlib inline
from matplotlib import pyplot as plt

def plot_data(data):
    plt.figure(figsize=(6, 4))
    trues = data[data[:,2] == True]
    plt.plot(trues[:,0], trues[:,1], 'bo', markersize=2)
    falses = data[data[:,2] == False]
    plt.plot(falses[:,0], falses[:,1], 'ro', markersize=2)
    plt.show()

def plot_decision_function(clf, X, y):
    h = .1  # step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(6, 4))
    plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)
    plt.title('Funcion de decision del modelo')
    plt.axis('tight')
    plt.show()
```

4.1. Gausianas con escaso solape - separables linealmente

Utilizo dos distribuciones gausianas de 2 dimensione con medias bien distintas.

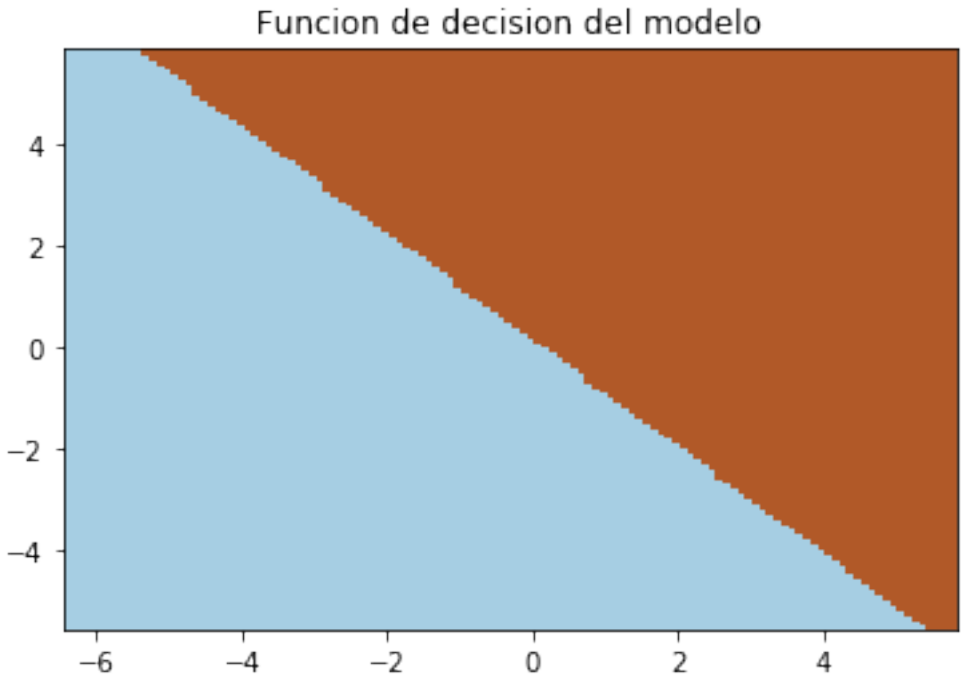
```
x = [[a,b,0] for a,b in np.random.normal(loc=-2, size=(1000,2))]
y = [[a,b,1] for a,b in np.random.normal(loc=2, size=(1000,2))]
data = np.concatenate([x,y])
np.random.shuffle(data)
X_train, y_train = data[:1500,:-1], data[:1500,-1]
X_test, y_test = data[1500:,:-1], data[1500:,-1]
plot_data(data)
```



Gráficamente sé ve que este ejemplo sí es linealmente separable. Vamos a confirmarlo encontrando una SVM con un buen rendimiento:

```
model = svm.SVC(kernel='linear').fit(X_train,y_train)
print 'Model C: %s' % model.C
print 'Model score: %s' % model.score(X_test,y_test)
plot_decision_function(model, X_train, y_train)
```

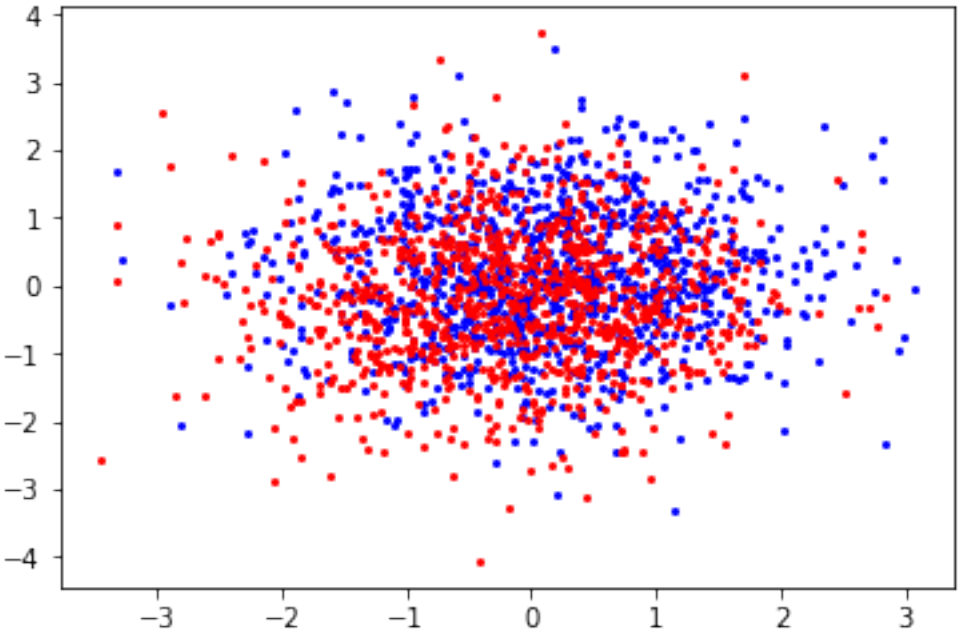
```
Model C: 1.0
Model score: 1.0
```



4.2. Gausianas fuertemente solapadas - no separables

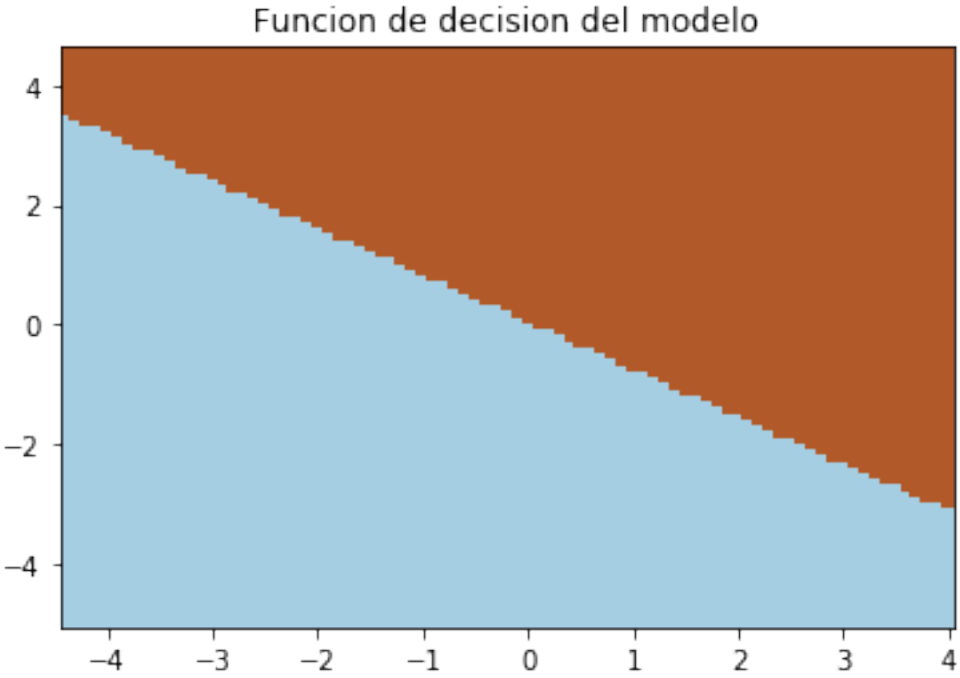
Utilizo dos distribuciones gausianas de 2 dimensione con medias no muy distintas.

```
x = [[a,b,0] for a,b in np.random.normal(loc=-0.2, size=(1000,2))]
y = [[a,b,1] for a,b in np.random.normal(loc=0.2, size=(1000,2))]
data = np.concatenate([x,y])
np.random.shuffle(data)
X_train, y_train = data[:1500,:-1], data[:1500,-1]
X_test, y_test = data[1500:,:-1], data[1500:,-1]
plot_data(data)
```




```
model = svm.SVC(kernel='linear').fit(X_train,y_train)
print 'Model C: %s' % model.C
print 'Model score: %s' % model.score(X_test,y_test)
plot_decision_function(model, X_train, y_train)
```

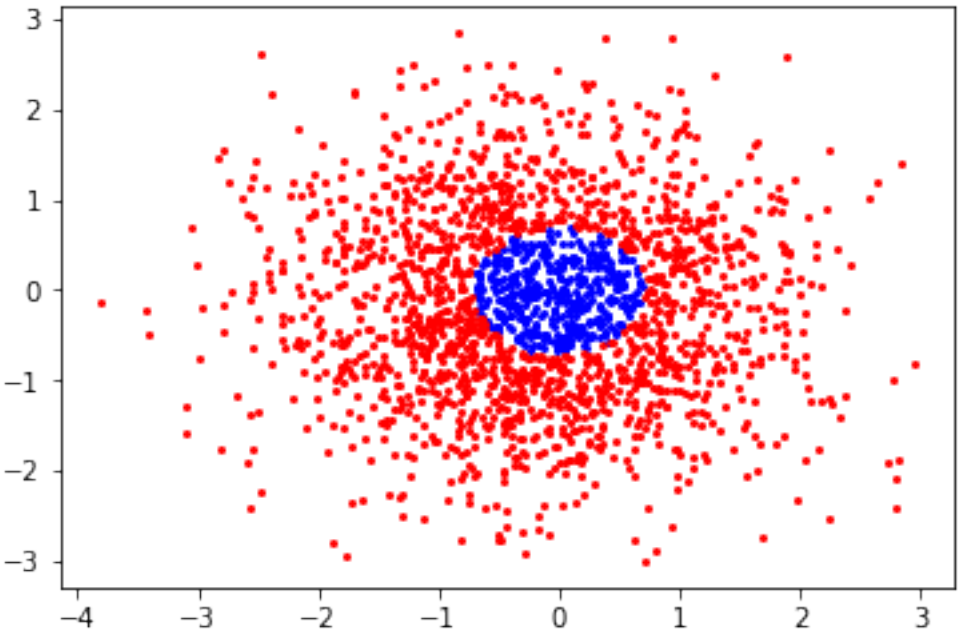
```
Model C: 1.0
Model score: 0.592
```



En este caso, el rendimiento del núcleo lineal ya no es bueno. No es de esperar que ningún núcleo obtenga buenos resultados de clasificación.

4.3. Gausianas separables polinómicamente

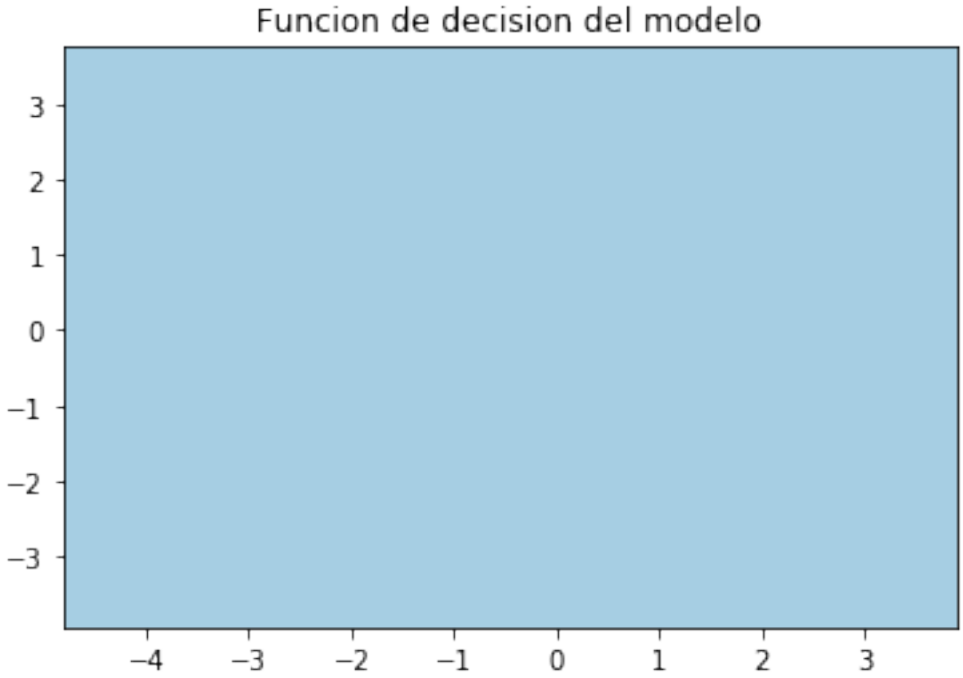
```
from math import sin
data = np.array([[a,b,a**2 + b**2 < 0.5] for a,b
                 in np.random.normal(loc=-0.2, size=(2000,2))])
X_train, y_train = data[:1500,:-1], data[:1500,-1]
X_test, y_test = data[1500:,:-1], data[1500:,-1]
plot_data(data)
```



El núcleo lineal parece ofrecer cierto scoring.

```
model = svm.SVC(kernel='linear').fit(X_train,y_train)
print 'Model C: %s' % model.C
print 'Model score: %s' % model.score(X_test,y_test)
plot_decision_function(model, X_train, y_train)
```

```
Model C: 1.0
Model score: 0.802
```

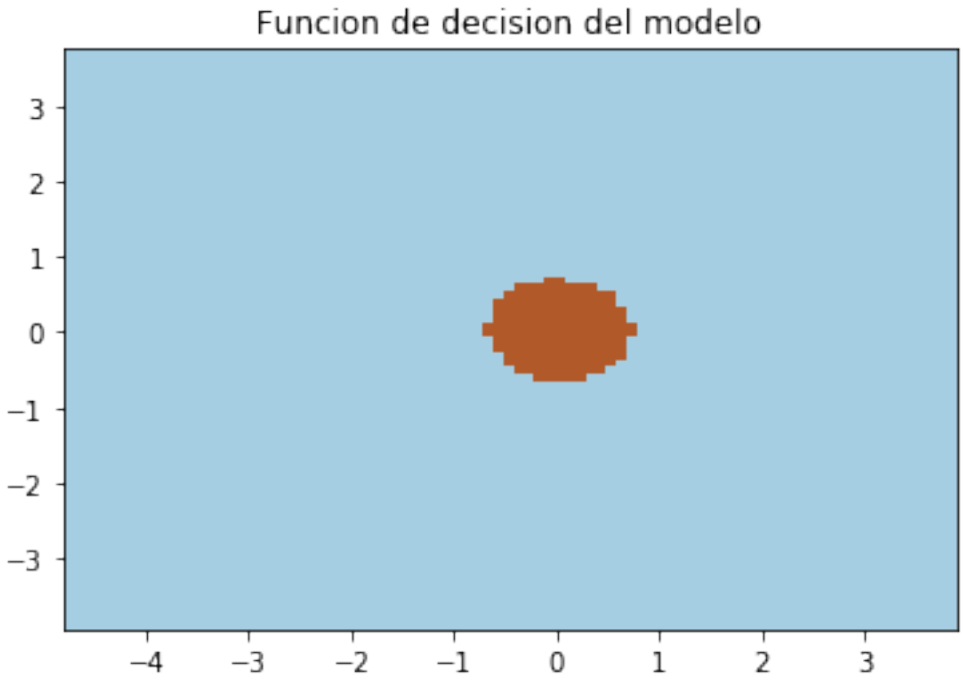


Sin embargo, lo que en realidad ocurre es que clasifica todas las intancias a la clase más frecuente en ellas (importante!). Así lo muestra la figura anterior.

En este problema, son los núcleos de base radial los que ofrecen el mejor resultado:

```
model = svm.SVC(kernel='rbf').fit(X_train,y_train)
print 'Model C: %s' % model.C
print 'Model score: %s' % model.score(X_test,y_test)
plot_decision_function(model, X_train, y_train)
```

```
Model C: 1.0
Model score: 0.994
```



5. Utilizar el resto de conjuntos y discutir los resultados

Aplico sin más el mismo procedimiento que hemos visto en los puntos anteriores.

Para simplificar la tarea, hacemos uso de unas funciones que cargan los datos de cada conjunto y realizan los distintos experimentos.

```
def prepare_data(name):
    train,meta = loadarff('svm-data/%s-train.arff' % name)
    test,_     = loadarff('svm-data/%s-test.arff' % name)
    top3_train,top3_test = top_three_categories(train,test)
    return normalize_data(top3_train,top3_test)
```

Y aplicamos los distintos clasificadores, con los parámetros por defecto para no eternizar la tarea.

```
def try_different_models(name):

    N = 20
    Cs = np.logspace(-3, 1, N)
    Gs = np.logspace(-3, 1, N)
    X_train,y_train,X_test,y_test = prepare_data(name)

    result = []
    gscv = GridSearchCV(estimator=svm.SVC(kernel='linear'),
                        param_grid=dict(C=Cs), n_jobs=-1)
    gscv.fit(X_train, y_train)
```

```
    result += [['Lineal', gscv.best_estimator_.C, None,
                gscv.best_estimator_.score(X_test, y_test)]]

    for n in range(1,5):
        gscv = GridSearchCV(estimator=svm.SVC(kernel='poly', degree=n),
                            param_grid=dict(C=Cs, gamma=Gs), n_jobs=-1)
        gscv.fit(X_train, y_train)
        result += [['Poly (grado %s)' % n,
                    gscv.best_estimator_.C, gscv.best_estimator_.gamma,
                    gscv.best_estimator_.score(X_test, y_test)]]

    gscv = GridSearchCV(estimator=svm.SVC(kernel='rbf'),
                        param_grid=dict(C=Cs, gamma=Gs), n_jobs=-1)
    gscv.fit(X_train, y_train)
    result += [['RBF', gscv.best_estimator_.C, gscv.best_estimator_.gamma,
                gscv.best_estimator_.score(X_test, y_test)]]

    gscv = GridSearchCV(estimator=svm.SVC(kernel='sigmoid'),
                        param_grid=dict(C=Cs,gamma=Gs), n_jobs=-1)
    gscv.fit(X_train, y_train)
    result += [['Sigmoid', gscv.best_estimator_.C, gscv.best_estimator_.gamma,
                gscv.best_estimator_.score(X_test, y_test)]]

    return result
```

5.1. Conjunto c

Definimos además una función que nos permite exportar a latex un dataframe de pandas.

```
import pandas as pd
pd.set_option('display.notebook_repr_html', True)

def _repr_latex_(self):
    return self.to_latex()

pd.DataFrame._repr_latex_ = _repr_latex_
```

Y realizamos el experimento:

```
result = try_different_models('c')
pd.DataFrame(data=result,columns=['kernel','C','gamma','scoring']).round(4)
```

	kernel	C	gamma	scoring
0	Lineal	10.0000	NaN	0.8655
1	Poly (grado 1)	1.4384	10.0000	0.8739
2	Poly (grado 2)	10.0000	10.0000	0.9244
3	Poly (grado 3)	6.1585	10.0000	0.9160
4	Poly (grado 4)	6.1585	10.0000	0.8992
5	RBF	10.0000	10.0000	0.9244
6	Sigmoid	2.3357	6.1585	0.8739

Fugura: resultados de probar distintios modelos en el conjunto c

La mejor puntuación la obtiene el núcleo rbf y el polinómico de grado 2. Sin embargo, uti-

lizan valores de C más altos que el sigmoid. Entiendo que la mejor opción sería el polinómico de grado 1, que no compremete el sobreajuste en realión al rendimiento.

5.2. Conjunto p

```
result = try_different_models('p')
pd.DataFrame(data=result,columns=['kernel','C','gamma','scoring']).round(4)
```

```
Top 3 classes: ['46', '20', '110']
```

	kernel	C	gamma	scoring
0	Lineal	10.0000	NaN	0.7983
1	Poly (grado 1)	6.1585	10.0	0.8571
2	Poly (grado 2)	3.7927	10.0	0.8571
3	Poly (grado 3)	3.7927	10.0	0.8487
4	Poly (grado 4)	3.7927	10.0	0.8487
5	RBF	6.1585	10.0	0.8487
6	Sigmoid	10.0000	10.0	0.8908

Fugura: resultados de probar distintios modelos en el conjunto p

Entiendo que una buena opción en este caso podría ser el núcleo polinómico de grado 2, puesto que ofrece buen resultado con penalización de error baja.

5.3. Conjunto v

```
result = try_different_models('v')
pd.DataFrame(data=result,columns=['kernel','C','gamma','scoring']).round(4)
```

```
Top 3 classes: ['46', '20', '110']
```

	kernel	C	gamma	scoring
0	Lineal	10.0000	NaN	0.7227
1	Poly (grado 1)	6.1585	10.0	0.8571
2	Poly (grado 2)	6.1585	10.0	0.8655
3	Poly (grado 3)	6.1585	10.0	0.8908
4	Poly (grado 4)	10.0000	10.0	0.8739
5	RBF	10.0000	10.0	0.8992
6	Sigmoid	10.0000	10.0	0.8487

Fugura: resultados de probar distintios modelos en el conjunto v

5.4. Conjunto h

```
result = try_different_models('h')
pd.DataFrame(data=result,columns=['kernel','C','gamma','scoring']).round(4)
```

```
Top 3 classes: ['46', '20', '110']
```

	kernel	C	gamma	scoring
0	Lineal	10.0000	NaN	0.6975
1	Poly (grado 1)	10.0000	10.0	0.8739
2	Poly (grado 2)	6.1585	10.0	0.8739
3	Poly (grado 3)	3.7927	10.0	0.8739
4	Poly (grado 4)	2.3357	10.0	0.8655
5	RBF	6.1585	10.0	0.8824
6	Sigmoid	10.0000	10.0	0.8655

Fugura: resultados de probar distintios modelos en el conjunto h

Entiendo que una buena opción en este caso podría ser el núcleo polinómico de grado 4 o el RBF, puesto que ofrecen buen resultado con penalización de error baja.

5.5. Conjunto z

```
result = try_different_models('z')
pd.DataFrame(data=result,columns=['kernel','C','gamma','scoring']).round(4)
```

```
Top 3 classes: ['46', '20', '110']
```

	kernel	C	gamma	scoring
0	Lineal	10.0000	NaN	0.6975
1	Poly (grado 1)	10.0000	10.0	0.8908
2	Poly (grado 2)	6.1585	10.0	0.8908
3	Poly (grado 3)	6.1585	10.0	0.8992
4	Poly (grado 4)	3.7927	10.0	0.8992
5	RBF	6.1585	10.0	0.8908
6	Sigmoid	10.0000	10.0	0.8824

Fugura: resultados de probar distintios modelos en el conjunto z

Entiendo que una buena opción en este caso podría ser el núcleo polinómico de grado 4, puesto que ofrece buen resultado con penalización de error baja.

5.6. Análisis de resultados en los conjuntos c, p, v, h y z

En ninguno de estos conjuntos el nucleo lineal ofrece resultados mejores que los otros núcleos, y siempre se encuentra con valores de C altos (10, el valor más alto considero en el experimento). Por eso me inclino a pensar que ninguno de estos conjuntos debe de ser linealmente separable.

En la sección anterior, para cada conjunto de datos, se han análizado los núcleos que podrían considerarse más convenientes.

6. Referencias

[1] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning, 20(3):273–297, 1995.*

[2] José Hernández Orallo and Mª José Ramírez Quintana. *Introducción a la Minería de Datos.*

[3] Qiang Wu and Ding-Xuan Zhou. Svm soft margin classifiers: linear programming versus quadratic programming. *Neural computation, 17(5):1160–1187, 2005.*

[4] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. *A Practical Guide to Support Vector Classification*