Minería de datos: máquinas de vectores soporte

Andrés Mañas Mañas

May 5, 2017

Contents

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.

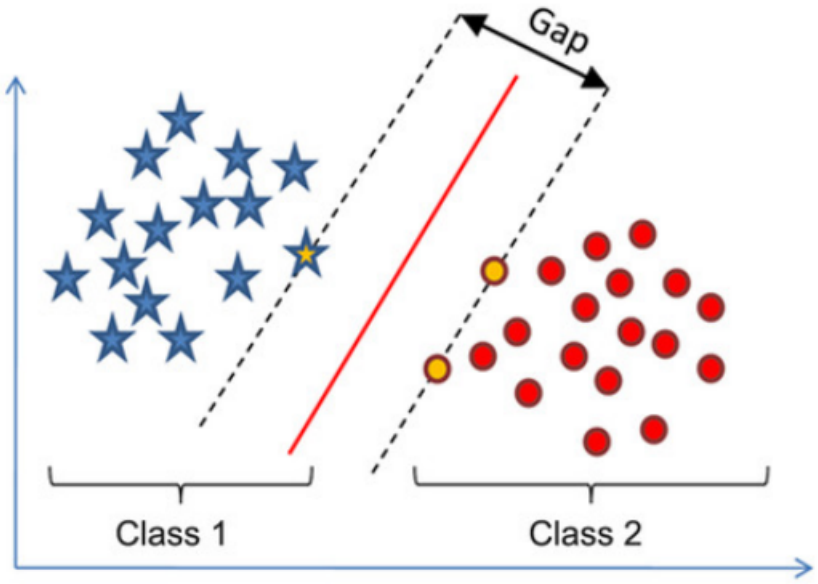


Figura 1: Ejemplo de 2 dimensiones para una máquina de soporte vectorial, obsérvese que la recta roja es la que maximiza la distancia entre los vectores de soporte (los puntos en amarillo)

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

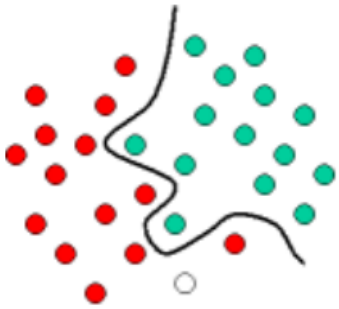


Figura 2: Espacio de 2 dimensiones en el que los patrones no son separables linealmente

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).

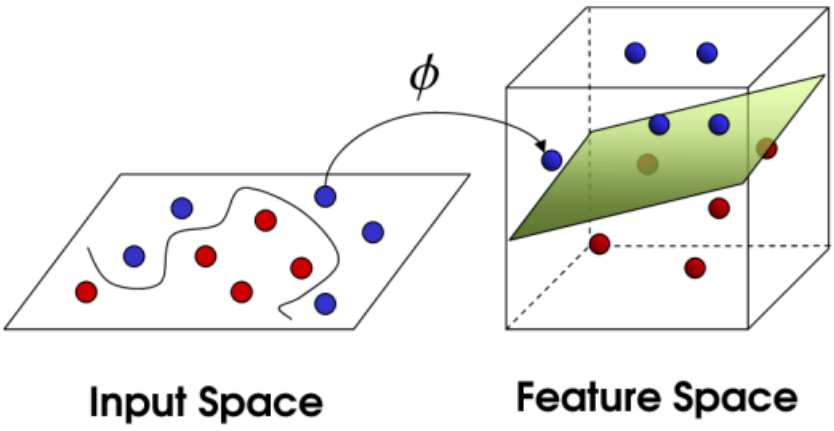


Figura 4: Kernel de 2 a 3 dimensiones dónde los patrones pueden ser perfectamente separados mediante un hiperplano

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.

Cuando se evalúan diferentes parámetros ("hiperparámetros") para los estimadores, como el ajuste **C** que debe configurarse manualmente para un SVM, todavía existe el riesgo de sobreajuste en el conjunto de prueba porque los parámetros pueden ajustarse hasta que el estimador se ejecute óptimamente. De esta forma, el conocimiento sobre el conjunto de pruebas puede **"filtrarse"** en el modelo y las métricas de evaluación ya no informan sobre el rendimiento de la generalización.

Para resolver este problema, otra parte del conjunto de datos puede ser presentada como un **"conjunto de validación"**: el entrenamiento continúa en el conjunto de entrenamiento, después de lo cual se realiza una evaluación en el conjunto de validación y cuando el experimento parece ser exitoso, la evaluación final se puede hacer en el conjunto de pruebas.

Sin embargo, al dividir los datos disponibles en tres conjuntos, reducimos drásticamente el número de muestras que se pueden utilizar para aprender el modelo, y los resultados pueden depender de una elección aleatoria particular para el par de conjuntos (entrenamiento, validación).

Una solución a este problema es un procedimiento llamado **validación cruzada** (CV para abreviar). Un conjunto de prueba todavía es necesario para la evaluación final, pero el conjunto de validación ya no es necesario al hacer CV. 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 "pliegues", en bucle:

- El modelo es entrenado usando k-1 de los pliegues como datos de entrenamiento
- El modelo resultante se valida en la parte restante de los datos

La medida de rendimiento reportada por la validación cruzada k-fold es el promedio de los valores calculados en el bucle. Este enfoque puede ser costoso desde el punto de vista computa-

cional, pero no desperdicia demasiados datos (como ocurre cuando se fija un conjunto de pruebas arbitrario), lo cual es una ventaja importante.

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 Seleccionar las 3 clases más frecuentes

```
import numpy as np

classes = np.concatenate((S_train['class'],S_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

S_top3_train = S_train[np.in1d(S_train['class'], top_three)]
S_top3_test  = S_test[np.in1d(S_test['class'], top_three)]
```

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

3.3 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

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

```
Cs: [  1.00000000e-03   1.62377674e-03   2.63665090e-03   4.28133240e-03
   6.95192796e-03   1.12883789e-02   1.83298071e-02   2.97635144e-02
   4.83293024e-02   7.84759970e-02   1.27427499e-01   2.06913808e-01
   3.35981829e-01   5.45559478e-01   8.85866790e-01   1.43844989e+00
   2.33572147e+00   3.79269019e+00   6.15848211e+00   1.00000000e+01]
```

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.

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 = S_top3_train[S_meta.names()[:-1]].reshape(-1, 1)
y_train = S_top3_train['class']

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: 0.001
Puntuación del mejor modelo: 0.753424657534
```

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

```
Cs = np.logspace(-5, 1, N)
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: 8.8586679041e-05
Puntuación del mejor modelo: 0.753424657534
```

Tomamos entonces como mejor valor **C = 8.8586679041e-05**.

Como la mejor máquina ofrece unos resultados de clasificación en torno al 0.75%, entiendo que el problema NO es linealmente separable. En mi opinión, los resultados del SVM con kernel lineal debieran ser mejores para considerar el problema linealmente separable.

Podemos evaluar entonces la mejor SVM encontrada en nuestro conjunto de prueba:

```
X_test = S_top3_test[S_meta.names()[:-1]].reshape(-1, 1)
y_test = S_top3_test['class']

print 'Puntuación en el conjunto de prueba: %s' % gscv.best_estimator_.score(X_test, y_test)
```

```
Puntuación en el conjunto de prueba: 0.680672268908
```

Por desgracia, en la evaluación de los datos de prueba el rendimiento se degrada un poco. Sin embargo, dado que el valor de C es muy bajo, no es de suponer se haya producido sobreajuste en el entrenamiento.

3.4 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.

```
gscv = GridSearchCV(estimator=svm.SVC(kernel='poly', degree=2),
                    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_
print 'Puntuación en el conjunto de prueba: %s' % gscv.best_estimator_.score(X_test, y_test)
```

```
Mejor C: 1e-05
Puntuación del mejor modelo: 0.72602739726
Puntuación en el conjunto de prueba: 0.663865546218
```

Vemos que el mejor resultado no mejora a los resultado obtenidos con un kernel lineal.

3.5 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. Disminuyo N a 10, es decir, 100 combinaciones en total.

```
N = 20
Cs = np.logspace(-3, 1, N)
Gs = np.logspace(-15, 3, N, base=2)
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 'Puntuación del mejor modelo: %s' % gscv.best_score_
print 'Puntuación en el conjunto de prueba: %s' % gscv.best_estimator_.score(X_test, y_test)
```

```
Mejor C: 0.0784759970351
Puntuación del mejor modelo: 0.753424657534
Puntuación en el conjunto de prueba: 0.663865546218
```

Observamos que los mejores resultados se encuentran con valores de gamma muy pequeños, aunque no mejoran los que se encontraron en el caso del núcleo lineal.

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

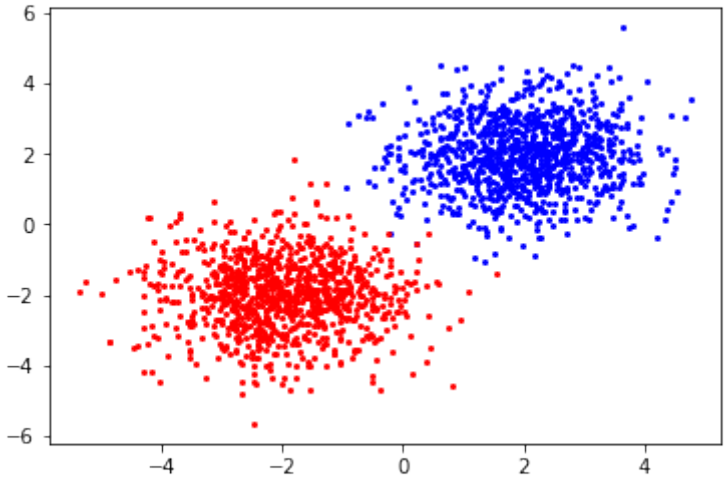
Definimos unas funciones que nos permita dibujar los datos y las curvas de decisión de los clasificadores obtenidos en cada experimento.

```
%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.

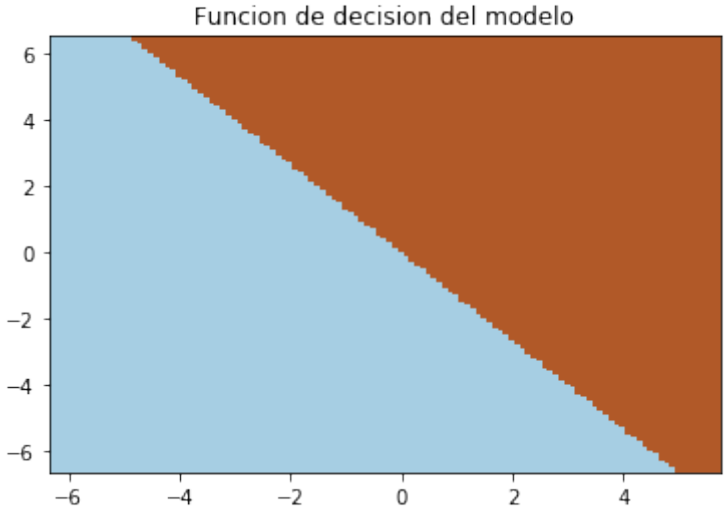
```
x = [[a,b,0] for a,b in np.random.normal(loc=-2, size=(1000,2))]
y = [[a,b,1] for a,b in np.random.normal(loc=2, size=(1000,2))]
data = np.concatenate([x,y])
np.random.shuffle(data)
X_train, y_train = data[:1500,:-1], data[:1500,-1]
X_test, y_test = data[1500:,:-1], data[1500:,-1]
plot_data(data)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



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 score: %s' % model.score(X_test,y_test)
plot_decision_function(model, X_train, y_train)
```

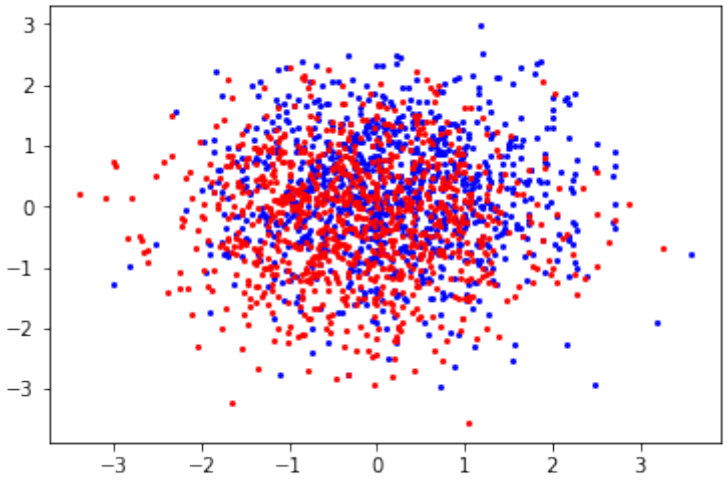
```
Model score: 0.994
```



4.2 Gausianas fuertemente solapadas - no separables

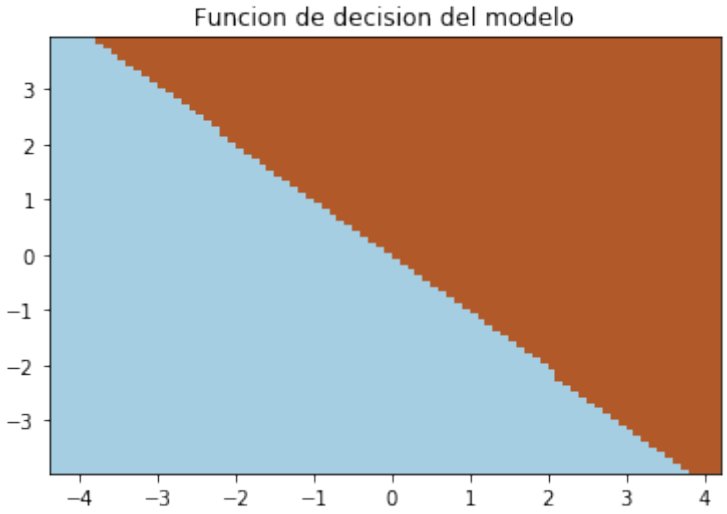
Utilizo dos distribuciones gausianas de 2 dimensione con medias no muy distintas.

```
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 score: %s' % model.score(X_test,y_test)
plot_decision_function(model, X_train, y_train)
```

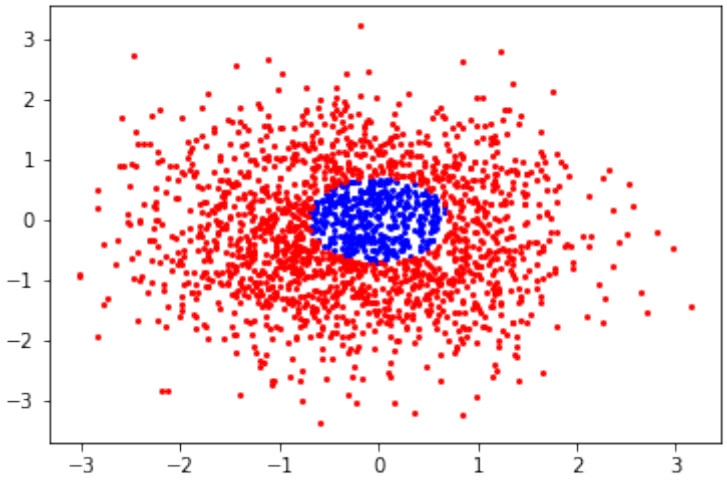
```
Model score: 0.616
```



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

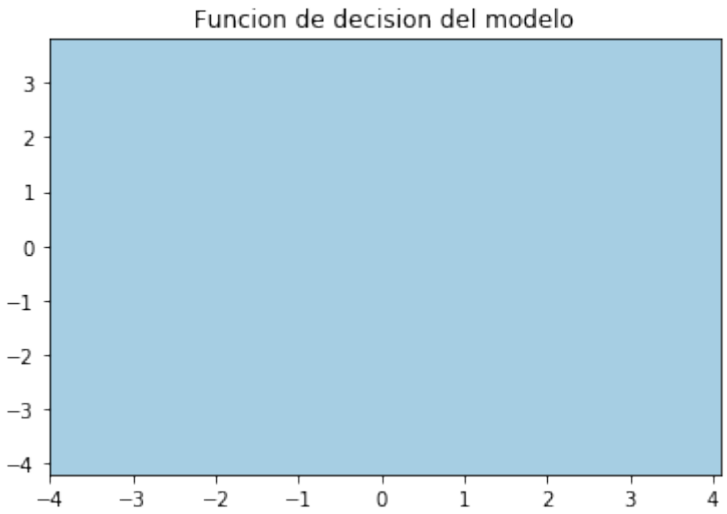
```
from math import sin
data = np.array([[a,b,a**2 + b**2 < 0.5] for a,b
                 in np.random.normal(loc=-0.2, size=(2000,2))])
X_train, y_train = data[:1500,:-1], data[:1500,-1]
X_test, y_test = data[1500:,:-1], data[1500:,-1]
plot_data(data)
```



El núcleo lineal parece ofrecer cierto scoring.

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

```
Model score: 0.802
```

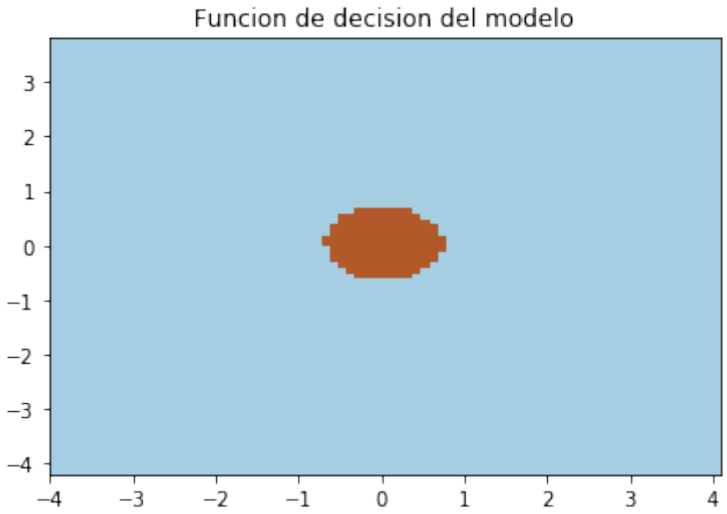


Sin embargo, lo que en realidad ocurre es que clasifica todas las intancias a la clase más frecuente en ellas (importante!). Así lo muestra la figura anterior.

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 score: %s' % model.score(X_test,y_test)
plot_decision_function(model, X_train, y_train)
```

```
Model score: 0.992
```



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)

    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]
    top3_train = train[np.in1d(train['class'], top_three)]
    top3_test  = test[np.in1d(test['class'], top_three)]

    X_train = top3_train[meta.names()[:-1]].reshape(-1, 1)
    y_train = top3_train['class']
    X_test =  top3_test[meta.names()[:-1]].reshape(-1, 1)
    y_test =  top3_test['class']
    return X_train, y_train, X_test, y_test
```

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

```
def try_different_models(X_train,y_train,X_test,y_test):
    model = svm.SVC(kernel='linear')
    model.fit(X_train, y_train)
    print 'Puntuación del clasificador lineal: %s' % model.score(X_test, y_test)

    model = svm.SVC(kernel='poly', degree=2)
    model.fit(X_train, y_train)
    print 'Puntuación del clasificador polinómico (grado 2): %s' \
            % model.score(X_test, y_test)

    model = svm.SVC(kernel='poly', degree=3)
    model.fit(X_train, y_train)
    print 'Puntuación del clasificador polinómico (grado 3): %s' \
            % model.score(X_test, y_test)

    model = svm.SVC(kernel='rbf')
    model.fit(X_train, y_train)
    print 'Puntuación del clasificador rbf: %s' % model.score(X_test, y_test)

    model = svm.SVC(kernel='sigmoid')
    model.fit(X_train, y_train)
    print 'Puntuación del clasificador sigmoid: %s' % model.score(X_test, y_test)
```

5.1 Conjunto c

```
X_train, y_train, X_test, y_test = prepare_data('c')
try_different_models(X_train, y_train, X_test, y_test)
```

```
Puntuación del clasificador lineal: 0.521008403361
Puntuación del clasificador polinómico (grado 2): 0.504201680672
Puntuación del clasificador polinómico (grado 3): 0.487394957983
Puntuación del clasificador rbf: 0.613445378151
```

```
Puntuación del clasificador sigmoid: 0.420168067227
```

La mejor puntuación la obtiene el núcleo rbf, aunque el resultado es bastante malo.

5.2 Conjunto p

```
X_train, y_train, X_test, y_test = prepare_data('p')
try_different_models(X_train, y_train, X_test, y_test)
```

```
Puntuación del clasificador lineal: 0.46218487395
Puntuación del clasificador polinómico (grado 2): 0.436974789916
Puntuación del clasificador polinómico (grado 3): 0.453781512605
Puntuación del clasificador rbf: 0.453781512605
Puntuación del clasificador sigmoid: 0.36974789916
```

5.3 Conjunto v

```
X_train, y_train, X_test, y_test = prepare_data('v')
try_different_models(X_train, y_train, X_test, y_test)
```

```
Puntuación del clasificador lineal: 0.495798319328
Puntuación del clasificador polinómico (grado 2): 0.512605042017
Puntuación del clasificador polinómico (grado 3): 0.504201680672
Puntuación del clasificador rbf: 0.512605042017
Puntuación del clasificador sigmoid: 0.36974789916
```

5.4 Conjunto h

```
X_train, y_train, X_test, y_test = prepare_data('h')
try_different_models(X_train, y_train, X_test, y_test)
```

```
Puntuación del clasificador lineal: 0.36974789916
Puntuación del clasificador polinómico (grado 2): 0.36974789916
Puntuación del clasificador polinómico (grado 3): 0.344537815126
Puntuación del clasificador rbf: 0.478991596639
Puntuación del clasificador sigmoid: 0.36974789916
```

5.5 Conjunto z

```
X_train, y_train, X_test, y_test = prepare_data('z')
try_different_models(X_train, y_train, X_test, y_test)
```

```
Puntuación del clasificador lineal: 0.386554621849
Puntuación del clasificador polinómico (grado 2): 0.319327731092
```

```
Puntuación del clasificador polinómico (grado 3): 0.352941176471
Puntuación del clasificador rbf: 0.470588235294
Puntuación del clasificador sigmoid: 0.36974789916
```

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

Comparados con los resultados que hemos obtenido para el conjunto s, estos resultados son bastante bajos. Sin embargo, en el caso del conjunto s llevamos a cabo una exploración bastante más profunda de los distintos núcleos y de los parámetros que se usaron en cada caso.

En el caso los conjuntos analizados en este apartado, se han tomado los parámetros por defecto de cada núcleo. Al no haberse hecho una búsqueda en el dominio de parámetros para dar con aquellos que mejor trabajen en cada problema es natural que los resultados no hayan sido satisfactorios.

Proceder a semejante análisis para cada uno de estos conjuntos se sale de la cantidad de tiempo que puedo dedicarle a esta práctica.

En cualquier caso, me parece importante resaltar que en 4 de los 5 conjuntos tratados en este apartado el núcleo basado en funciones de base radial ha ofrecido el mejor rendimiento. Esto está en coherencia con lo que citaba en el punto 1.5, dónde según [4] el núcleo RBF suele ser un buen comienzo en la exploración.

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*