Project 2: Supervised Learning

Building a Student Intervention System

1. Classification vs Regression

Your goal is to identify students who might need early intervention - which type of supervised machine learning problem is this, classification or regression? Why?

Problem type

The output variable "passed" is nominal with two categories: "no" and "yes". This is therefore a classification problem

2. Exploring the Data

Let's go ahead and read in the student dataset first.

*To execute a code cell, click inside it and press **Shift+Enter**.*

In [24]:
```
# Import libraries
import numpy as np
import pandas as pd
```

In [25]:
```
# Read student data
student_data = pd.read_csv("student-data.csv")
print "Student data read successfully!"
# Note: The last column 'passed' is the target/label, all other are
feature columns
```

```
Student data read successfully!
```

Now, can you find out the following facts about the dataset?

- Total number of students
- Number of students who passed
- Number of students who failed
- Graduation rate of the class (%)
- Number of features

*Use the code block below to compute these values. Instructions/steps are marked using **TODO**s.*

In [26]:
```
# TODO: Compute desired values - replace each '?' with an appropriat
e expression/function call
n_students = len(student_data)
n_features = len(student_data.columns)-1
n_passed =  len(student_data.passed[student_data.passed=='yes'])
n_failed =  len(student_data.passed[student_data.passed=='no'])
grad_rate = 100.0*n_passed/n_students
print "Total number of students: {}".format(n_students)
print "Number of students who passed: {}".format(n_passed)
print "Number of students who failed: {}".format(n_failed)
print "Number of features: {}".format(n_features)
print "Graduation rate of the class: {:.2f}%".format(grad_rate)
```

```
Total number of students: 395
Number of students who passed: 265
Number of students who failed: 130
Number of features: 30
Graduation rate of the class: 67.09%
```

3. Preparing the Data

In this section, we will prepare the data for modeling, training and testing.

Identify feature and target columns

It is often the case that the data you obtain contains non-numeric features. This can be a problem, as most machine learning algorithms expect numeric data to perform computations with.

Let's first separate our data into feature and target columns, and see if any features are non-numeric.
Note: For this dataset, the last column (`'passed'`) is the target or label we are trying to predict.

In [27]:
```
# Extract feature (X) and target (y) columns
feature_cols = list(student_data.columns[:-1])  # all columns but la
st are features
target_col = student_data.columns[-1]  # last column is the target/l
abel
print "Feature column(s):-\n{}".format(feature_cols)
print "Target column: {}".format(target_col)

X_all = student_data[feature_cols]  # feature values for all student
s
y_all = student_data[target_col]  # corresponding targets/labels
print "\nFeature values:-"
print X_all.head()  # print the first 5 rows
```

```
Feature column(s):-
['school', 'sex', 'age', 'address', 'famsize', 'Pstatus', 'Medu',
'Fedu', 'Mjob', 'Fjob', 'reason', 'guardian', 'traveltime', 'study
time', 'failures', 'schoolsup', 'famsup', 'paid', 'activities', 'n
ursery', 'higher', 'internet', 'romantic', 'famrel', 'freetime', '
goout', 'Dalc', 'Walc', 'health', 'absences']
Target column: passed

Feature values:-
  school sex  age address famsize Pstatus  Medu  Fedu     Mjob
  Fjob  \
0     GP   F   18       U     GT3       A     4     4  at_home   t
eacher
1     GP   F   17       U     GT3       T     1     1  at_home
 other
2     GP   F   15       U     LE3       T     1     1  at_home
 other
3     GP   F   15       U     GT3       T     4     2   health  se
rvices
4     GP   F   16       U     GT3       T     3     3    other
 other

    ...    higher internet  romantic  famrel  freetime goout Dalc
Walc health  \
0   ...       yes       no        no       4         3     4    1
   1      3
1   ...       yes      yes        no       5         3     3    1
   1      3
2   ...       yes      yes        no       4         3     2    2
   3      3
3   ...       yes      yes       yes       3         2     2    1
   1      5
4   ...       yes       no        no       4         3     2    1
   2      5

  absences
0        6
1        4
2       10
3        2
4        4

[5 rows x 30 columns]
```

Preprocess feature columns

As you can see, there are several non-numeric columns that need to be converted! Many of them are simply `yes`/`no`, e.g. `internet`. These can be reasonably converted into `1`/`0` (binary) values.

Other columns, like `Mjob` and `Fjob`, have more than two values, and are known as *categorical variables*. The recommended way to handle such a column is to create as many columns as possible values (e.g. `Fjob_teacher`, `Fjob_other`, `Fjob_services`, etc.), and assign a `1` to one of them and `0` to all others.

These generated columns are sometimes called *dummy variables*, and we will use the [`pandas.get_dummies()` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html?highlight=get_dummies#pandas.get_dummies)](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html?highlight=get_dummies#pandas.get_dummies) function to perform this transformation.

In [28]:

```
# Preprocess feature columns
def preprocess_features(X):
    outX = pd.DataFrame(index=X.index)  # output dataframe, initiall
y empty

    # Check each column
    for col, col_data in X.iteritems():
        # If data type is non-numeric, try to replace all yes/no val
ues with 1/0
        if col_data.dtype == object:
            col_data = col_data.replace(['yes', 'no'], [1, 0])
        # Note: This should change the data type for yes/no columns
to int

        # If still non-numeric, convert to one or more dummy variabl
es
        if col_data.dtype == object:
            col_data = pd.get_dummies(col_data, prefix=col)  # e.g.
'school' => 'school_GP', 'school_MS'

        outX = outX.join(col_data)  # collect column(s) in output da
taframe

    return outX

X_all = preprocess_features(X_all)
print "Processed feature columns ({}):-\n{}".format(len(X_all.column
s), list(X_all.columns))
```

```
Processed feature columns (48):-
['school_GP', 'school_MS', 'sex_F', 'sex_M', 'age', 'address_R', '
address_U', 'famsize_GT3', 'famsize_LE3', 'Pstatus_A', 'Pstatus_T'
, 'Medu', 'Fedu', 'Mjob_at_home', 'Mjob_health', 'Mjob_other', 'Mj
ob_services', 'Mjob_teacher', 'Fjob_at_home', 'Fjob_health', 'Fjob
_other', 'Fjob_services', 'Fjob_teacher', 'reason_course', 'reason
_home', 'reason_other', 'reason_reputation', 'guardian_father', 'g
uardian_mother', 'guardian_other', 'traveltime', 'studytime', 'fai
lures', 'schoolsup', 'famsup', 'paid', 'activities', 'nursery', 'h
igher', 'internet', 'romantic', 'famrel', 'freetime', 'goout', 'Da
lc', 'Walc', 'health', 'absences']
```

Split data into training and test sets

So far, we have converted all *categorical* features into numeric values. In this next step, we split the data (both features and corresponding labels) into training and test sets.

In [29]:
```
# First, decide how many training vs test samples you want
num_all = student_data.shape[0]  # same as len(student_data)
num_train = 300  # about 75% of the data
num_test = num_all - num_train

from sklearn.cross_validation import train_test_split
# TODO: Then, select features (X) and corresponding labels (y) for t
he training and test sets
# Note: Shuffle the data or randomly select samples to avoid any bia
s due to ordering in the dataset
#X_train = ?
#y_train = ?
#X_test = ?
#y_test = ?
X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, te
st_size=1.0*num_test/num_all, random_state=42)
print "Training set: {} samples".format(X_train.shape[0])
print "Test set: {} samples".format(X_test.shape[0])
# Note: If you need a validation set, extract it from within trainin
g data
```

```
Training set: 300 samples
Test set: 95 samples
```

4. Training and Evaluating Models

Choose 3 supervised learning models that are available in scikit-learn, and appropriate for this problem. For each model:

- What are the general applications of this model? What are its strengths and weaknesses?
- Given what you know about the data so far, why did you choose this model to apply?
- Fit this model to the training data, try to predict labels (for both training and test sets), and measure the F_1 score. Repeat this process with different training set sizes (100, 200, 300), keeping test set constant.

Produce a table showing training time, prediction time, F_1 score on training set and F_1 score on test set, for each training set size.

Note: You need to produce 3 such tables - one for each model.

Model selection

Many of the predictor variables (features) are binary or factor variables. After preprocessing we have 48 features. We have 300 samples for training. Not much samples tor train a model with 48 features. Therefore a model shall not have to much parameters to be trained. Futhermore the decision boundary is probably very complex and nonlinear. Therefore the model shall only require a small amount of training data and shall be able to handle complex decision boundary.

For this reasons I used the following models:

- Naive Bayes
- K-Neighbours
- Support Vector Machines with RBF-Kernel

In [30]:

```
# Train a model
import time

def train_classifier(clf, X_train, y_train):
    print "Training {}...".format(clf.__class__.__name__)
    start = time.time()
    clf.fit(X_train, y_train)
    end = time.time()
    print "Done!\nTraining time (secs): {:.3f}".format(end - start)

# TODO: Choose a model, import it and instantiate an object
from sklearn.naive_bayes import GaussianNB

clf = GaussianNB()

# Fit model to training data
train_classifier(clf, X_train, y_train)  # note: using entire traini
ng set here
#print clf  # you can inspect the learned model by printing it
```

```
Training GaussianNB...
Done!
Training time (secs): 0.001
```

In [31]:
```
# Predict on training set and compute F1 score
from sklearn.metrics import f1_score

def predict_labels(clf, features, target):
    print "Predicting labels using {}...".format(clf.__class__.__nam
e__)
    start = time.time()
    y_pred = clf.predict(features)
    end = time.time()
    print "Done!\nPrediction time (secs): {:.3f}".format(end - start
)
    return f1_score(target.values, y_pred, pos_label='yes')

train_f1_score = predict_labels(clf, X_train, y_train)
print "F1 score for training set: {}".format(train_f1_score)
```

```
Predicting labels using GaussianNB...
Done!
Prediction time (secs): 0.000
F1 score for training set: 0.80378250591
```

In [32]:
```
# Predict on test data
print "F1 score for test set: {}".format(predict_labels(clf, X_test,
 y_test))
```

```
Predicting labels using GaussianNB...
Done!
Prediction time (secs): 0.001
F1 score for test set: 0.763358778626
```

Classifier I: Naive Bayes

As first classifier I use Gaussian Naive Bayes. The likelihood if the features is assumed to be Gaussian.

Advantages of Naive Bayes:

- very efficient for training, in terms of the total number of computations needed
- able to cope with complex decision boundaries.
- works well with small training samples.
- performs well on many different training tasks, especially for text classification.

Disadvantage of Naive Bayes:

- each features are considered independently and therefore naive Bayes will be inpropriate if the strong conditional independence assumption is violated.

In [33]:
```
# Train and predict using different training set sizes
def train_predict(clf, X_train, y_train, X_test, y_test):
    print "------------------------------------------"
    print "Training set size: {}".format(len(X_train))
    train_classifier(clf, X_train, y_train)
    print "F1 score for training set: {}".format(predict_labels(clf,
 X_train, y_train))
    print "F1 score for test set: {}".format(predict_labels(clf, X_t
est, y_test))

# TODO: Run the helper function above for desired subsets of trainin
g data
# Note: Keep the test set constant
clf=GaussianNB()
train_predict(clf, X_train[:100], y_train[0:100], X_test, y_test)
train_predict(clf, X_train[:200], y_train[0:200], X_test, y_test)
train_predict(clf, X_train[:300], y_train[0:300], X_test, y_test)
```

```
------------------------------------------
Training set size: 100
Training GaussianNB...
Done!
Training time (secs): 0.000
Predicting labels using GaussianNB...
Done!
Prediction time (secs): 0.000
F1 score for training set: 0.846715328467
Predicting labels using GaussianNB...
Done!
Prediction time (secs): 0.000
F1 score for test set: 0.802919708029
------------------------------------------
Training set size: 200
Training GaussianNB...
Done!
Training time (secs): 0.000
Predicting labels using GaussianNB...
Done!
Prediction time (secs): 0.001
F1 score for training set: 0.840579710145
Predicting labels using GaussianNB...
Done!
Prediction time (secs): 0.001
F1 score for test set: 0.724409448819
------------------------------------------
Training set size: 300
Training GaussianNB...
Done!
Training time (secs): 0.001
Predicting labels using GaussianNB...
Done!
Prediction time (secs): 0.000
F1 score for training set: 0.80378250591
Predicting labels using GaussianNB...
Done!
Prediction time (secs): 0.000
F1 score for test set: 0.763358778626
```

Results of Naive Bayes

In [34]:
```
import pandas as pd
import numpy as np
rowNames=["100","200","300"]
colNames=["Training time (secs)", "Testing time (secs)","F1 Score fo
r training set", "F1 Score for test set"]
data = np.array([[0.001, 0.000, 0.001],
                 [0.000, 0.000, 0.000],
                 [0.846715328467, 0.840579710145, 0.80378250591],
                [0.802919708029, 0.724409448819, 0.763358778626]])

rows=["A","B","C"]
df=pd.DataFrame(data,colNames,rowNames)
print "                              Traing set size"
print df
```

```
                          Traing set size
                               100       200       300
Training time (secs)      0.001000  0.000000  0.001000
Testing time (secs)       0.000000  0.000000  0.000000
F1 Score for training set 0.846715  0.840580  0.803783
F1 Score for test set     0.802920  0.724409  0.763359
```

As we can see, the Naive Bayes performrs quite well and it has very small training and prediction times.

Second classifier: K-Neighbors.

The K-Neighbors-Classifier is a quite simple classifier. It is a parameterless method to estimate density distributions.

Advantages:

- simple and can cope with complicated decision boundaries.
- training is fast.

Disadvantages:

- memory and runtime consumption for high dimensional problems ist high. Since our problem at hand is small, this will not be an issue.

Note: the number of classes must be given in advance. Here we will use the default of 5 classes.

In [35]:
```
from sklearn.neighbors import KNeighborsClassifier

clf=KNeighborsClassifier()

train_predict(clf, X_train[:100], y_train[0:100], X_test, y_test)
train_predict(clf, X_train[:200], y_train[0:200], X_test, y_test)
train_predict(clf, X_train[:300], y_train[0:300], X_test, y_test)
```

```
------------------------------------------
Training set size: 100
Training KNeighborsClassifier...
Done!
Training time (secs): 0.001
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.001
F1 score for training set: 0.805970149254
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.001
F1 score for test set: 0.724637681159
------------------------------------------
Training set size: 200
Training KNeighborsClassifier...
Done!
Training time (secs): 0.001
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.002
F1 score for training set: 0.88
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.001
F1 score for test set: 0.769230769231
------------------------------------------
Training set size: 300
Training KNeighborsClassifier...
Done!
Training time (secs): 0.000
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.005
F1 score for training set: 0.880898876404
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.002
F1 score for test set: 0.780141843972
```

Results for K-Neigbors

K-Neigbors perfors much better than Naive Bayes. Training and testing time is very low.

In [36]:
```
rowNames=["100","200","300"]
colNames=["Training time (secs)", "Testing time (secs)","F1 Score fo
r training set", "F1 Score for test set"]
data = np.array([[0.000, 0.000, 0.001],
                 [0.001, 0.001, 0.002],
                 [0.805970149254, 0.88, 0.880898876404],
                [0.724637681159, 0.769230769231, 0.780141843972]])

rows=["A","B","C"]
df=pd.DataFrame(data,colNames,rowNames)
print "                              Traing set size"
print df
```

```
                           Traing set size
                                100       200       300
Training time (secs)       0.000000  0.000000  0.001000
Testing time (secs)        0.001000  0.001000  0.002000
F1 Score for training set  0.805970  0.880000  0.880899
F1 Score for test set      0.724638  0.769231  0.780142
```

Third classifier: Support Vector Machine (SVM)

Support vector machines are very powerfull classifiers.

Advantages:

- they can cope with very complex decision boundaries.
- effective in high dimensional spaces.
- still effective in cases where number fo dimensions is greater than the number of samples.
- Versatile: they can be adapted to the given problem by using different kernels.
- Uses a subset of training points in the decision function (support vectors), so it is memory efficient.

Disadvantages

- their compute and storage requirements increase rapidly with the number of training vectors.
- If the number of features is much greater than the number of samples, the method is likely to give poor performances.
- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation

Here we use the SVM classifier with default parameters, e.g with RBF as kernel.

In [37]:
```
from sklearn.svm import SVC
clf=SVC()
train_predict(clf, X_train[:100], y_train[0:100], X_test, y_test)
train_predict(clf, X_train[:200], y_train[0:200], X_test, y_test)
train_predict(clf, X_train[:300], y_train[0:300], X_test, y_test)
```

```
------------------------------------------
Training set size: 100
Training SVC...
Done!
Training time (secs): 0.002
Predicting labels using SVC...
Done!
Prediction time (secs): 0.000
F1 score for training set: 0.877697841727
Predicting labels using SVC...
Done!
Prediction time (secs): 0.000
F1 score for test set: 0.774647887324
------------------------------------------
Training set size: 200
Training SVC...
Done!
Training time (secs): 0.003
Predicting labels using SVC...
Done!
Prediction time (secs): 0.001
F1 score for training set: 0.867924528302
Predicting labels using SVC...
Done!
Prediction time (secs): 0.001
F1 score for test set: 0.781456953642
------------------------------------------
Training set size: 300
Training SVC...
Done!
Training time (secs): 0.005
Predicting labels using SVC...
Done!
Prediction time (secs): 0.004
F1 score for training set: 0.876068376068
Predicting labels using SVC...
Done!
Prediction time (secs): 0.001
F1 score for test set: 0.783783783784
```

Results for SVM

The f1 metrics for the test set are 0.774 training size 100), 0.781 (training size 200) and 0.784 (training size 300). These results are better than for Naive Bayes and better than for KNeighbors. Run time is the highest of all tested classifiers.

In [38]:
```
rowNames=["100","200","300"]
colNames=["Training time (secs)", "Testing time (secs)","F1 Score fo
r training set", "F1 Score for test set"]
data = np.array([[0.001, 0.002, 0.005],
                 [0.001, 0.001, 0.001],
                 [0.877697841727, 0.867924528302, 0.876068376068],
                [0.774647887324, 0.781456953642,0.783783783784]])

rows=["A","B","C"]
df=pd.DataFrame(data,colNames,rowNames)
print "                                Traing set size"
print df
```

```
                           Traing set size
                                100       200       300
Training time (secs)       0.001000  0.002000  0.005000
Testing time (secs)        0.001000  0.001000  0.001000
F1 Score for training set  0.877698  0.867925  0.876068
F1 Score for test set      0.774648  0.781457  0.783784
```

5. Choosing the Best Model

- Based on the experiments you performed earlier, in 1-2 paragraphs explain to the board of supervisors what single model you chose as the best model. Which model is generally the most appropriate based on the available data, limited resources, cost, and performance?
- In 1-2 paragraphs explain to the board of supervisors in layman's terms how the final model chosen is supposed to work (for example if you chose a Decision Tree or Support Vector Machine, how does it make a prediction).
- Fine-tune the model. Use Gridsearch with at least one important parameter tuned and with at least 3 settings. Use the entire training set for this.
- What is the model's final F_1 score?

Results of the experiments

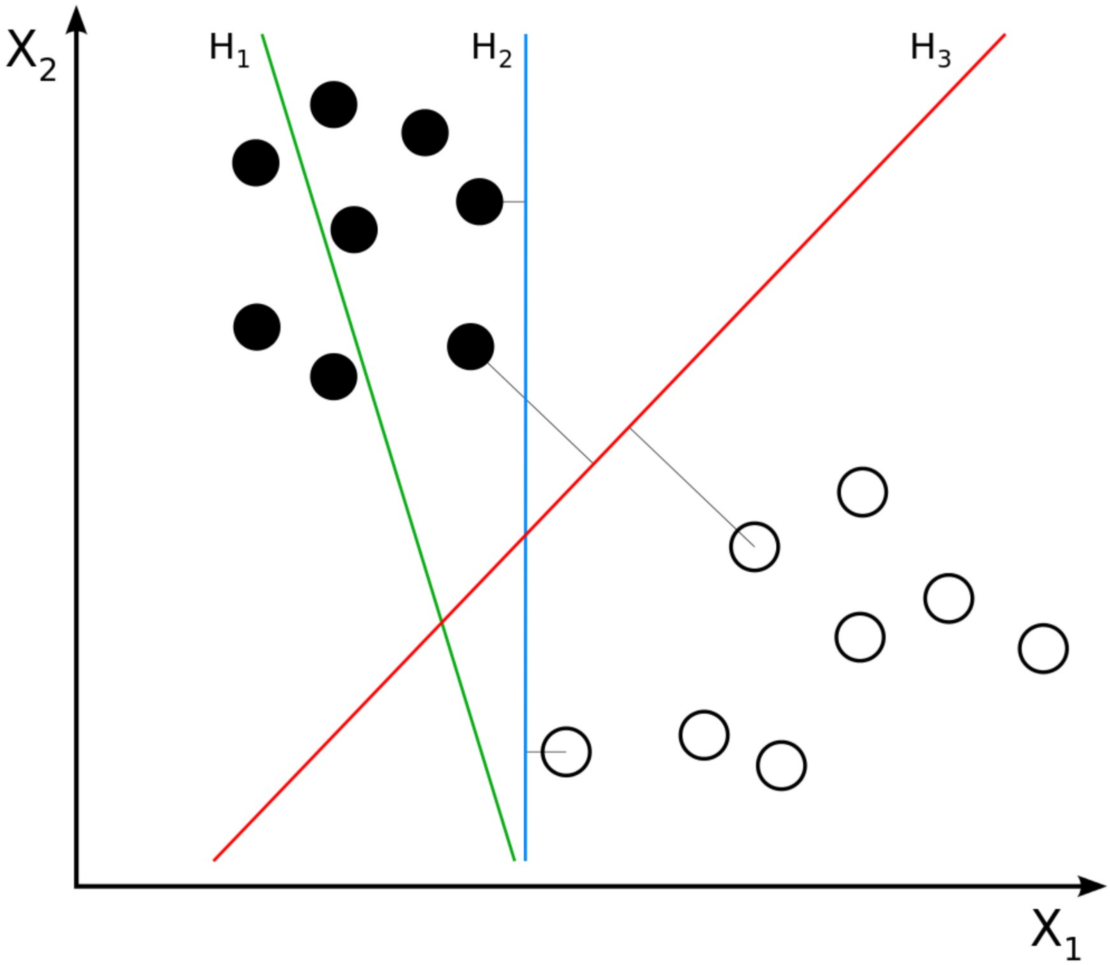
We evaluate three different types of classifiers: Naive Bayes, KNeighors and Support Vector Machines (SVM). I will not dive deep into the technical details of these classifiers. The best results we get for SVM. The runtime consumption for naive Bayes is very low, but it gives the worsest prediction. Kneighbors is quite fast for our problem and it gives good results, but SVM gives the best results. SVM has the highest runtime , but our problem size is small and the runtime to predict the results for 100 students is about 1 ms. This is fast enough for our purposes.

We choose therefore SVM classifier. This classifier is rather powerful and flexible. The SVM works as follows:

- A support vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks. A good separation is achieved by the hyperplane that has the largest distance to the nearest training-data point of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier. The goal is therefore to maximzis this margin. The following figure shows different planes H1, h2 and H3 to separate the white and the black points. H1 does not separate this both classes, H2 does but with a low margin, h3 seperates them with the maximum margin.

In [39]:
```
from IPython.display import Image
Image(url='https://upload.wikimedia.org/wikipedia/commons/thumb/b/b5
/Svm_separating_hyperplanes_%28SVG%29.svg/1000px-Svm_separating_hype
rplanes_%28SVG%29.svg.png')
```

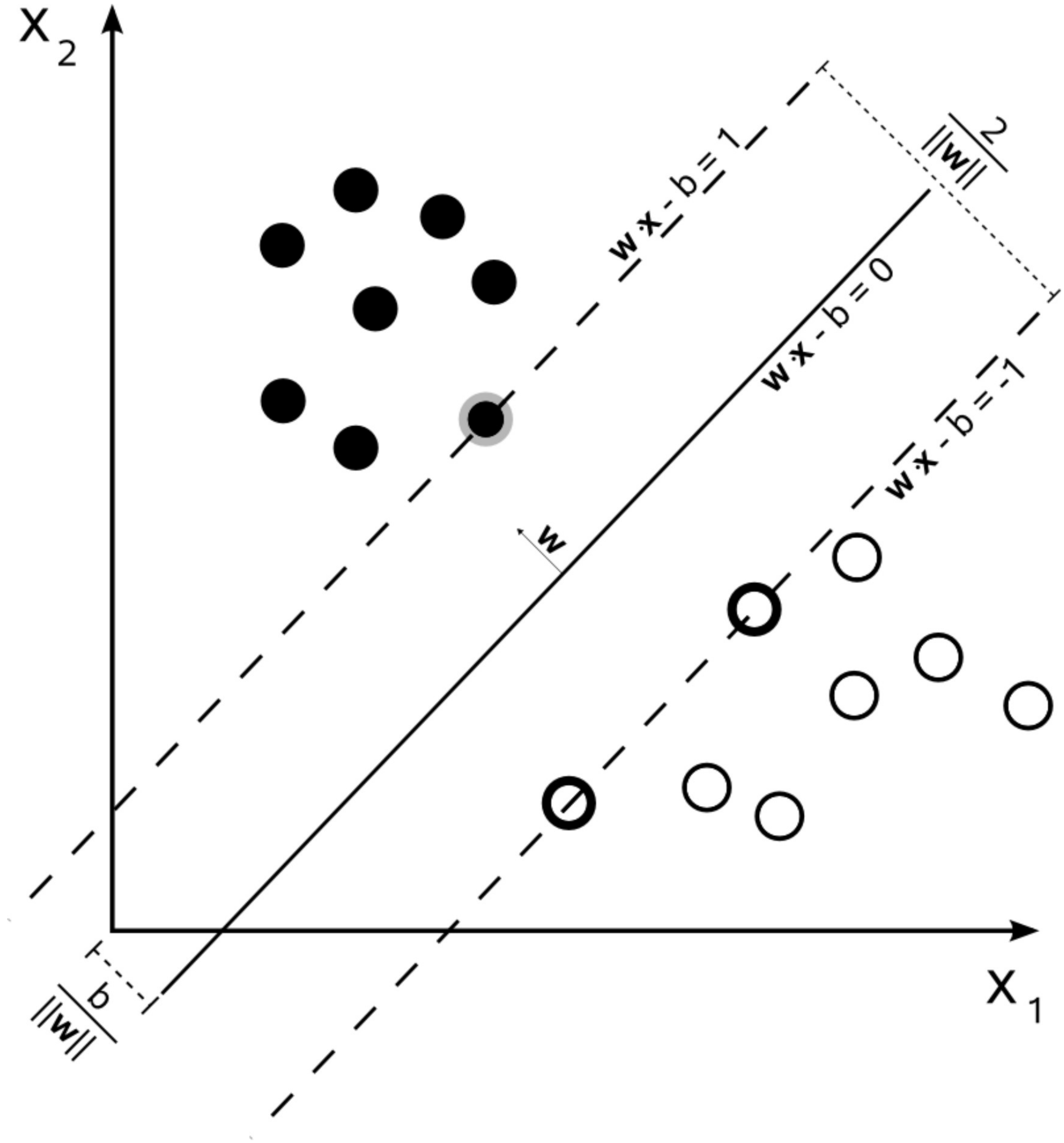
Out[39]:



- The next figure shows the maximum-margin hyperplane and margins for an SVM trained with samples from two classes. Samples on the margin are called the support vectors. This gives this model its name: support vector machine. This vectors are important, because after training only those vectors contribute to the parameters of the hyperplane.

In [40]:
```
Image(url='https://upload.wikimedia.org/wikipedia/commons/2/2a/Svm_max_sep_hyperplane_with_margin.png')
```

Out[40]:

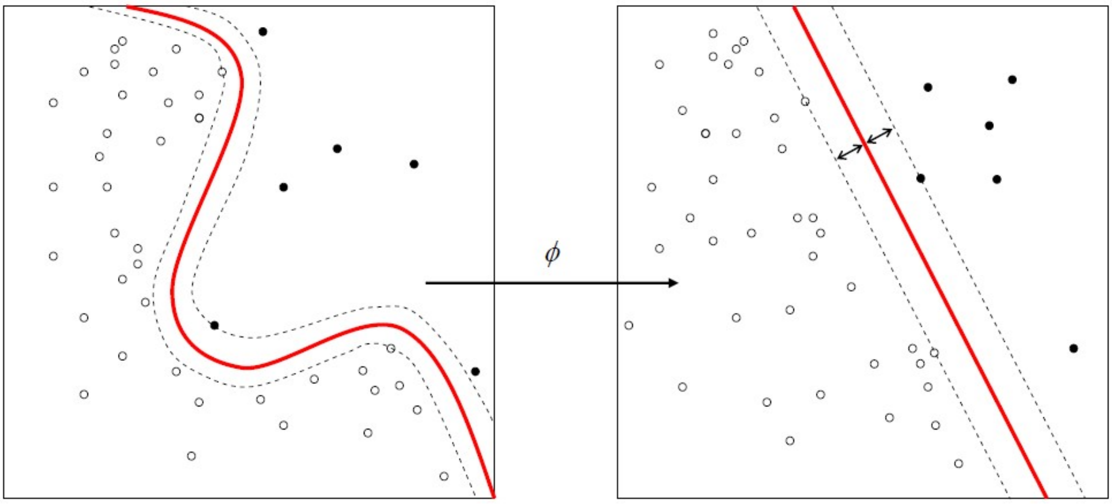


- During training the separating hyperplanes are calculated. Prediction is made by calculating on which sides of the hpyerplanes a point is. This defines the class of the point.

In real world problems groups of different points are seldom separable by linear planes. Here a very clever and powerful trick is the so called "kernel trick". This means instead of a linear plane we use a nonlinear function - a kernel function - to separate two classes. A kernel function simply transform a curved decision boundary to a linear plane as shown in the following figure.

In [41]:
```
Image(url='https://upload.wikimedia.org/wikipedia/commons/1/1b/Kerne
l_Machine.png')
```

Out[41]:



Model tuning

Gridsearch is used for tuning. We use the following parameters for tuning: gamma, C and the kernel.

In [42]:
```
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import f1_score
from sklearn.metrics import make_scorer
f1_scorer = make_scorer(f1_score, pos_label="yes")
clf=SVC()

parameters={'kernel':('linear', 'rbf', 'sigmoid'), 'C':[0.1, 1, 10,
100], 'gamma':[0.001, 0.1, 10, 100]}
clfGS = GridSearchCV(clf,parameters,cv=10,scoring=f1_scorer)
train_predict(clfGS, X_train[:300], y_train[0:300], X_test, y_test)
print clfGS.best_params_
```

```
------------------------------------------
Training set size: 300
Training GridSearchCV...
Done!
Training time (secs): 131.039
Predicting labels using GridSearchCV...
Done!
Prediction time (secs): 0.005
F1 score for training set: 0.97619047619
Predicting labels using GridSearchCV...
Done!
Prediction time (secs): 0.002
F1 score for test set: 0.789473684211
{'kernel': 'rbf', 'C': 1, 'gamma': 0.1}
```

Apply Gridsearch now on a subset of the parameters around the best found. Note: Kernel rbf is the default kernel.

In [43]:
```
parameters={'gamma':np.linspace(0.01,0.2,10),'C':np.linspace(0.5,1.5
,10)}
clfGS = GridSearchCV(clf,parameters,cv=10,scoring=f1_scorer)
train_predict(clfGS, X_train[:300], y_train[0:300], X_test, y_test)
print clfGS.best_params_
```

```
------------------------------------------
Training set size: 300
Training GridSearchCV...
Done!
Training time (secs): 8.246
Predicting labels using GridSearchCV...
Done!
Prediction time (secs): 0.005
F1 score for training set: 0.990338164251
Predicting labels using GridSearchCV...
Done!
Prediction time (secs): 0.002
F1 score for test set: 0.794701986755
{'C': 1.2777777777777777, 'gamma': 0.094444444444444442}
```

Results of tuned SVM

F1 0.7947 with RBF kernel, C= 1.27777 and gamma=0.09444