student_intervention

April 29, 2016

1 Project 2: Supervised Learning

1.0.1 Building a Student Intervention System

1.1 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?

It's a classification problem because the model output will basically be a binary value that answers the question: is the student likely to pass the exam or not?

1.2 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 [78]: # Import libraries
         import numpy as np
         import pandas as pd
```

```
In [79]: # 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 [80]: # TODO: Compute desired values - replace each '?' with an appropriate expression/function call
         n_students = len(student_data)
         n_features = len(student_data.columns) - 1 # last column is the target/label
         n_passed = len(student_data[(student_data['passed'] == 'yes')])
         n_failed = len(student_data[(student_data['passed'] == 'no')])
         grad_rate = (n_passed * 1.0)/ n_students *100
         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%
```

1.3 3. Preparing the Data

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

1.3.1 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 [81]: # Extract feature (X) and target (y) columns
         feature_cols = list(student_data.columns[:-1])  # all columns but last are features
         target_col = student_data.columns[-1]  # last column is the target/label
         print "Feature column(s):-\n{}".format(feature_cols)
         print "Target column: {}".format(target_col)

         X_all = student_data[feature_cols]  # feature values for all students
         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', 'gu
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   teacher
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  services
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]
```

1.3.2 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()` function to perform this transformation.

```
In [82]: # Preprocess feature columns
         def preprocess_features(X):
             outX = pd.DataFrame(index=X.index)  # output dataframe, initially empty

             # Check each column
             for col, col_data in X.iteritems():
                 # If data type is non-numeric, try to replace all yes/no values 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 variables
                 if col_data.dtype == object:
                     col_data = pd.get_dummies(col_data, prefix=col)  # e.g. 'school' => 'school_GP', '

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

             return outX

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

1.3.3 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 [83]: # 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

         # TODO: Then, select features (X) and corresponding labels (y) for the training and test sets\
         # Note: Shuffle the data or randomly select samples to avoid any bias due to ordering in the d

         from sklearn.cross_validation import train_test_split

         X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=95, random_state=1

         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 training data"
```

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

1.4 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 F1 score. Repeat this process with different training set sizes (100, 200, 300), keeping test set constant.

Produce a table showing training time, prediction time, F1 score on training set and F1 score on test set, for each training set size.

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

Random Forests: Random Forests are an ensemble learning method, they combine the predictions of different learning models in order in order to improve generalizability and robustness over a single estimator. In particular they fit a user specified number of tree classifiers on different subsets of the training data. It then use average to improve accuracy and control over-fitting. This procedure improve model performance because it decreases the variance of the model with little or no increase to the bias. The size of the sub-samples is the same of the original input, what changes are the features that are taken into account: at each split a different subset of features is picked up by the algorithm and one of them is selected. The split is then performed on that feature. This procedure is done in order to limit the development of correlation among different trees: this would likely happen as usually there are few features that are strong predictors and all the trees would end up doing the same split on those. Despite usually having to train a large number of trees RF are very fast to train as the fitting alghorithm can be easily parallelized by having one tree being trained on independent hardware. Thanks to this property RF run well on large datasets. RF as well as trees are non-parametric which means that both models make no assumption on the distribution of the data. This property gives the RF the ability to describe many more distributions than other models: linear and kernel regression, for example, are limited in descrybing distributions that fit well with the function that has been chosen as core. Another advantage of using RF is that, thanks to the entropy analisys, it estimates which variables are important by preferring splitting data according to those features that generates more knowledge. The main con is that classifications made by RFs are too complicated to be interpreted by humans(not true for a single tree).

In this case given that data are few and have a lot of features I figured that RF would perform well as they are able to determine which features have a stronger prediction power and they immediately perform split on those. Also, given the possibility to parallelize the tasks I expect them to be quick in training which will be one of the things that I'll need to take into account when making a decision on the classifier. Lastly they are not parametric which is good since I have no idea how a figure in 48 dimesion look like. I specified the depth to 8 as I figured that 2^8 = 256 is a sufficient number of leaves to obtain a good characterization of the data without overfitting. I also specified the size of the forest to 128 trees as this paper argue (https://www.researchgate.net/publication/230766603_How_Many_Trees_in_a_Random_Forest) should be enough to reduce the variance of the model as much as possible.

Source: Course material https://www.researchgate.net/publication/230766603_How_Many_Trees_in_a_Random_Forest http://scikit-learn.org/stable/modules/ensemble.html https://en.wikipedia.org/wiki/Random_forest

K Nearest Neighbor: Is a type of instance based learning which chacarcterize those models that instead of trying to generalize a distribution into a formula or object, compares new data with instances saved in memory during training. KNN in particular consists in taking the K (user defined variable) data closest to the instance to predict and calulate some sort of average (weighted or unweighted mean or user defined) of their labels. Grater K smoother the resulting boundary. KNN is a non parametric method and as such makes little assumptions about the data distribution: smoothness, data belonging to nearby regions of space also belong to the same class, and that data belong to a space where the distance actually means something in terms of belonging. Metrics different from the Euclidean can be defined for specific data, like categorical. Because it takes into account numerous instances in order to predict a value or a class this algorith is robust against noisy training data. Training of KNN is very fast as it simply consists in storing the training data, predicting, on the other hand, is computationally expensive because it needs to calculate

retrieve the K closest instances and calculate their mean, in case of continous data, or have them vote for the class to predict. Another drawback is that all the features are taken into account with the same importance, while this might not be necessarily true. Also, as the number of features linearly increases, the amount of data that we need in order to properly generalize grows exponentially.

I chose to consider this model because it is non parametric, very fast in learning and retains all the information of the data. Even though it needs all the data to be stored, the size of the dataset is so small that I feel it to be a non factor. K = 17 because I read online that a good K is usually close to the squared root of the number of samples.

Source: Course material http://people.revoledu.com/kardi/tutorial/KNN/Strength%20and%20Weakness.htm https://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Classification/kNN http://scikit-learn.org/stable/modules/neighbors.html

SVM with RBF Kernel: SVMs are a class of methods that learn by determining and maximizing the separation margin among two classes. The power of SVMs is that through the definition of different kernel functions they are able to describe data with different distributions: from the simplest linearly separable to others more complicated (polynomially separable, rbf...). Basically what we are doing by specifying a non linear kernel is to transform the data in a higher dimensional space where they are linearly separable, maximize the linear margin and then trasform it back to the original space obtaining the nonlienar margin outline. SVMs are parametric methods because of the strong assumptions that are made on data distribution and the goal of learning parameters that describe a separation function.
Although they can be computationally heavy during fitting (training time is cubicly proportional to the number of data) they are extremely fast in testing as the result of the training are a subset of support vectors that refer to those points that most influence the margin. SVMs perform extremely well in cases where data can be sharply separated, independently from the type of space where this is true (as long as we can define a Kernel that describes the similarity of the data). They also perform well in high dimensional spaces, even those where the number of dimensions is higher than those of the data. However performance drops when the classes tend to overlap and when the number of features is greater than those of samples.
Another con is that there is no multi-class SVM, to obtain so there's the need to combine two or more SVMs.

After having a look online I chose the rbf kernel because of this paper (http://jmlr.org/papers/volume15/delgado14a/delgado14a.pdf) and of a user suggestion on stack-overflow that says that for small amount of data with a number of features that is relatively large compared to it these are usually those that perform better: "...if your feature number is small (10^0 - 10^3), and the sample number is intermediate (10^1 - 10^4), use Gaussian kernel will be better." (http://stackoverflow.com/questions/20566869/where-is-it-best-to-use-svm-with-linear-kernel).

Source: Course material http://stackoverflow.com/questions/20566869/where-is-it-best-to-use-svm-with-linear-kernel http://jmlr.org/papers/volume15/delgado14a/delgado14a.pdf http://www.nickgillian.com/wiki/pmwiki.php/GRT/SVM http://scikit-learn.org/stable/modules/svm.html

1.4.1 Random Forest

In [84]:
```
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

# train a random forest

from sklearn.ensemble import RandomForestClassifier
```

```
        clf_rfc = RandomForestClassifier(n_estimators = 128, n_jobs = 4)

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

```
Training RandomForestClassifier...
Done!
Training time (secs): 0.546
```

```
In [85]: # 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__.__name__)
            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_rfc, X_train, y_train)
        print "F1 score for training set: {}".format(train_f1_score)
```

```
Predicting labels using RandomForestClassifier...
Done!
Prediction time (secs): 0.208
F1 score for training set: 1.0
```

```
In [86]: # Predict on test data
        print "F1 score for test set: {}".format(predict_labels(clf_rfc, X_test, y_test))
```

```
Predicting labels using RandomForestClassifier...
Done!
Prediction time (secs): 0.233
F1 score for test set: 0.857142857143
```

```
In [87]: # 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_test, y_test))

        # TODO: Run the helper function above for desired subsets of training data
        # Note: Keep the test set constant

        for i in [1,2,3]:
            train_predict(clf_rfc, X_train[: i*100], y_train[: i*100], X_test, y_test)
```

```
------------------------------------------
Training set size: 100
Training RandomForestClassifier...
Done!
Training time (secs): 0.485
```

```
Predicting labels using RandomForestClassifier...
Done!
Prediction time (secs): 0.247
F1 score for training set: 1.0
Predicting labels using RandomForestClassifier...
Done!
Prediction time (secs): 0.252
F1 score for test set: 0.797202797203
------------------------------------------
Training set size: 200
Training RandomForestClassifier...
Done!
Training time (secs): 0.485
Predicting labels using RandomForestClassifier...
Done!
Prediction time (secs): 0.247
F1 score for training set: 1.0
Predicting labels using RandomForestClassifier...
Done!
Prediction time (secs): 0.250
F1 score for test set: 0.851612903226
------------------------------------------
Training set size: 300
Training RandomForestClassifier...
Done!
Training time (secs): 0.492
Predicting labels using RandomForestClassifier...
Done!
Prediction time (secs): 0.217
F1 score for training set: 1.0
Predicting labels using RandomForestClassifier...
Done!
Prediction time (secs): 0.207
F1 score for test set: 0.837837837838
```

Random Forest Classifier

Training size	Training time	Prediction time	F1 training	F1 test
100	0.485	0.252	1.0	0.797202797203
200	0.485	0.250	1.0	0.851612903226
300	0.492	0.207	1.0	0.837837837838

1.4.2 KNN

In [43]:
```
# train a Nearest Neighbor Classifier

from sklearn.neighbors import KNeighborsClassifier

clf_neigh = KNeighborsClassifier(n_neighbors = 17, weights = 'distance')

# Fit model to training data
train_classifier(clf_neigh, X_train, y_train)
```

```
Training KNeighborsClassifier...
```

```
Done!
Training time (secs): 0.002
```

```
In [44]: # Predict on training set and compute F1 score
         train_f1_score = predict_labels(clf_neigh, X_train, y_train)
         print "F1 score for training set: {}".format(train_f1_score)
```

```
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.008
F1 score for training set: 1.0
```

```
In [45]: # Predict on test data
         print "F1 score for test set: {}".format(predict_labels(clf_neigh, X_test, y_test))
```

```
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.003
F1 score for test set: 0.857142857143
```

```
In [46]: for i in [1,2,3]:
             train_predict(clf_neigh, X_train[: i*100], y_train[: i*100], 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: 1.0
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.001
F1 score for test set: 0.849315068493
----------------------------------------
Training set size: 200
Training KNeighborsClassifier...
Done!
Training time (secs): 0.001
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.004
F1 score for training set: 1.0
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.002
F1 score for test set: 0.868421052632
----------------------------------------
Training set size: 300
Training KNeighborsClassifier...
Done!
Training time (secs): 0.001
Predicting labels using KNeighborsClassifier...
```

```
Done!
Prediction time (secs): 0.008
F1 score for training set: 1.0
Predicting labels using KNeighborsClassifier...
Done!
Prediction time (secs): 0.003
F1 score for test set: 0.857142857143
```

K Nearest Neighbor Classifier

Training size	Training time	Prediction time	F1 training	F1 test
100	0.001	0.001	1.0	0.849315068493
200	0.001	0.002	1.0	0.868421052632
300	0.001	0.003	1.0	0.857142857143

1.4.3 SVC with RBF Kernel

In [67]:
```
# train a SVM rbf  kernel

from sklearn.svm import SVC

clf_svc = SVC(kernel = 'rbf')

# Fit model to training data
train_classifier(clf_svc, X_train, y_train)
```

```
Training SVC...
Done!
Training time (secs): 0.007
```

In [68]:
```
# Predict on training set and compute F1 score
train_f1_score = predict_labels(clf_svc, X_train, y_train)
print "F1 score for training set: {}".format(train_f1_score)
```

```
Predicting labels using SVC...
Done!
Prediction time (secs): 0.005
F1 score for training set: 0.858387799564
```

In [69]:
```
# Predict on test data
print "F1 score for test set: {}".format(predict_labels(clf_svc, X_test, y_test))
```

```
Predicting labels using SVC...
Done!
Prediction time (secs): 0.002
F1 score for test set: 0.846153846154
```

In [70]:
```
for i in [1,2,3]:
    train_predict(clf_svc, X_train[: i*100], y_train[: i*100], X_test, y_test)
```

```
------------------------------------------
Training set size: 100
Training SVC...
Done!
Training time (secs): 0.001
```

```
Predicting labels using SVC...
Done!
Prediction time (secs): 0.001
F1 score for training set: 0.859060402685
Predicting labels using SVC...
Done!
Prediction time (secs): 0.000
F1 score for test set: 0.833333333333
------------------------------------------
Training set size: 200
Training SVC...
Done!
Training time (secs): 0.003
Predicting labels using SVC...
Done!
Prediction time (secs): 0.002
F1 score for training set: 0.858064516129
Predicting labels using SVC...
Done!
Prediction time (secs): 0.001
F1 score for test set: 0.84076433121
------------------------------------------
Training set size: 300
Training SVC...
Done!
Training time (secs): 0.006
Predicting labels using SVC...
Done!
Prediction time (secs): 0.005
F1 score for training set: 0.858387799564
Predicting labels using SVC...
Done!
Prediction time (secs): 0.002
F1 score for test set: 0.846153846154
```

Support Vector Classifier with RBF Kernel

Training size	Training time	Prediction time	F1 training	F1 test
100	0.001	0.000	0.859060402685	0.833333333333
200	0.004	0.001	0.858064516129	0.84076433121
300	0.007	0.002	0.858387799564	0.846153846154

1.5 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 F1 score?

Random forest: Given the experiments performed earlier Random Forests (RF) with 128 trees perform well in classifying the data with an F1 score consistently greater than 0.82, (0.8 is the result for a model that predicts all passes, our baseline). However performance time both in fitting and prediction is in the order of tenth of seconds, two orders of magnitude greater than the other two classifiers I chose. When I tried to run the fitting in parallel the performance time increased of 0.15s on average: I think that it is caused by the initialization of the data on different cores. Most likely with these little data this action is not worth the effort. Despite the good performance the time needed to train and predict is considerably larger than the other two models, thus I don't think it is the best model for this situation.

K-Nearest Neighbor: As expected KNN is faster in training than both RT and SVC while in testing is comparable with SVCs. Score performance is good, around 0.81, but is still lower than SVC score. Overall I think that this model might only worth to be considered only if we had many more data, and only if the cost of training them was really high and higher than the storage cost. And even in this case I feel that it might not be worth to choose KNN as training is something that is normally done once (unless data change continously) while prediction is done more often. It might be the case that you still prefer a model that takes longer to train but is more precise and comparably fast in predicting rather than one fast in training.

SVC with RBF Kernel: The performance time of this model is comparable with KNN, although slightly slower in training, moreover its F1 score is the highest among the three. Normally these models don't perform well because they have a running time that scales quadratically with the number of instances to train. In this case however the dataser is small and this ends up not being an issue. Thus, I think that for this problem SVM with RBF kernel is the best model to use.

SVMs Layman explanation: An instance of a set of data when plotted is a point in a space that has as many dimensions as the features of data. Classification aims to find rules to tell which characteristics make a point belong to one class or another. SVMs represent a classification method that wants to determine the linear boundary (best margin) that best divides the data. Said line (or plane) is the one that is the most distant from the nearest points belonging to the two classes. Those points that influence the margin are called support vectors. In Figure 1 there is a graphical representation of linear separable data, those points that act as support vectors and the best margin.

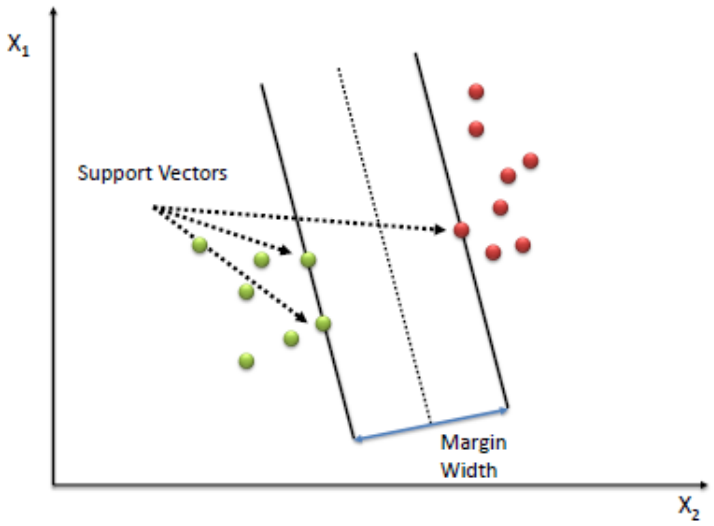

Figure 1: linear svm

Sometimes however it is not possible to draw a line that separates the data because of how the features constraints them. The solution in these cases is to project(draw them with a new set of coordinates depending on the original ones) the points into a different space where features assume simpler values and thus can be divided with a plane. This process is extremely clear in Figure 2: as shown on the right data cannot be separated by a line in a two dimensional space, however by projecting them on a three dimentional space (left) they are now easily separable by a plane. This is called the kernel trick and can be applied to a lot of

different space transformations. Another neat property of data projection is that once the margin has been identified it is possible to invert the transformation and now the margin separating the two classes will be a non linear one but will still separate the data correctly. Again, in the figure on the right the green ring is the plane (the margin on the left) projected in a lower dimentional space.

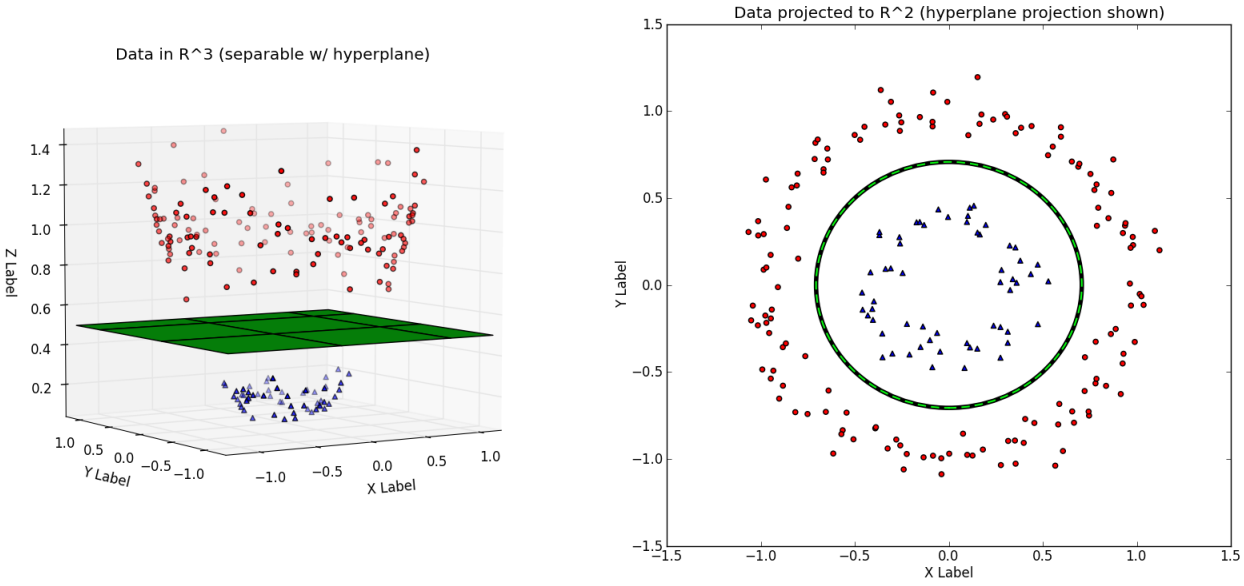

Figure 2: linearly married

F1 score: After normalizing the training data and tuning both gamma and C the final f1 score on the test set of this classifier is 0.846153846154

```
In [33]: # TODO: Fine-tune your model and report the best F1 score

In [77]: from sklearn.metrics import make_scorer
         from sklearn import grid_search

         classifier = SVC(kernel = 'rbf')

         parameters = {'C':(0.2, 0.4, 0.6, 0.8, 1.0), 'gamma':(0.0000001, 0.005, 0.009, 0.021, 0.07, 0.

         scoring_function = make_scorer(f1_score, greater_is_better = True, pos_label = 'yes')

         clf = grid_search.GridSearchCV(classifier, parameters, scoring_function, refit = True)

         clf.fit(X_train, y_train)

         print "F1 score for test set: {}".format(predict_labels(clf, X_test, y_test))

         clf.best_params_

Predicting labels using GridSearchCV...
Done!
Prediction time (secs): 0.002
F1 score for test set: 0.846153846154

Out[77]: {'C': 1.0, 'gamma': 0.07}

In [ ]:
```