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?

This is a classification problem. We need to give a students a label: needs early intervention or doesn't need early intervention. Those problems in which there are a finite number of possible outputs and they cannot be ordered are classification problems. Regression problems involve continuous outputs, or, at least, ordinal outputs.

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 [1]:

```
# Import libraries
import numpy as np
import pandas as pd
from sklearn.cross_validation import train_test_split
import random
```

In [2]:

```
# Set random seed for reproducibility.
random.seed(9)

# Read student data
student_data = pd.read_csv("student-data.csv")
print "Student data read successfully!"
```

```
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

In [3]:

```
n_students = len(student_data)
n_features = len(student_data.columns) - 1
n_passed = sum(student_data['passed'] == 'yes')
n_failed = sum(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 [4]:

```
# Extract feature (X) and target (y) columns
feature_cols = list(student_data.columns[:-1])  # all columns but last are featu
res
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', 'F
edu', 'Mjob', 'Fjob', 'reason', 'guardian', 'traveltime', 'studytime
', 'failures', 'schoolsup', 'famsup', 'paid', 'activities', 'nursery
', '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   tea
cher
1     GP   F   17       U     GT3       T     1     1  at_home     o
ther
2     GP   F   15       U     LE3       T     1     1  at_home     o
ther
3     GP   F   15       U     GT3       T     4     2   health  serv
ices
4     GP   F   16       U     GT3       T     3     3    other     o
ther

    ...    higher internet  romantic  famrel  freetime goout Dalc Wa
lc 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) function to perform this transformation.

In [5]:

```
# 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', 'school_MS'

        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_a
ll.columns))
```

```
Processed feature columns (48):-
['school_GP', 'school_MS', 'sex_F', 'sex_M', 'age', 'address_R', 'ad
dress_U', 'famsize_GT3', 'famsize_LE3', 'Pstatus_A', 'Pstatus_T', 'M
edu', 'Fedu', 'Mjob_at_home', 'Mjob_health', 'Mjob_other', 'Mjob_ser
vices', 'Mjob_teacher', 'Fjob_at_home', 'Fjob_health', 'Fjob_other',
'Fjob_services', 'Fjob_teacher', 'reason_course', 'reason_home', 're
ason_other', 'reason_reputation', 'guardian_father', 'guardian_mothe
r', 'guardian_other', 'traveltime', 'studytime', 'failures', 'school
sup', 'famsup', 'paid', 'activities', 'nursery', 'higher', 'internet
', 'romantic', 'famrel', 'freetime', 'goout', 'Dalc', 'Walc', 'healt
h', '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 [6]:

```
# 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

# Then, select features (X) and corresponding labels (y) for the training and te
st sets
# Note: Shuffle the data or randomly select samples to avoid any bias due to ord
ering in the dataset
X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=num_
test)
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
```

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.

In [7]:

```
import time

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

    return training_time
```

```
from sklearn.metrics import f1_score

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

def train_predict(clf, X_train, y_train, X_test, y_test, verbose):
    if verbose:
        print "------------------------------------------"
        print "Training set size: {}".format(len(X_train))
    # Average over several runs as there can be a lot of variability.  It might
be good to use cross-validation.
    num_trials = 64
    total_training_time = 0
    total_training_set_score = 0
    total_training_prediction_time = 0
    total_test_set_score = 0
    total_test_prediction_time = 0
    for i in range(num_trials):
        training_time = train_classifier(clf, X_train, y_train, verbose)
        total_training_time += training_time

        training_set_score, training_prediction_time = predict_labels(clf, X_tra
in, y_train, verbose)
        total_training_set_score += training_set_score
        total_training_prediction_time += training_prediction_time

        test_set_score, test_prediction_time = predict_labels(clf, X_test, y_tes
t, verbose)
        total_test_set_score += test_set_score
        total_test_prediction_time += test_prediction_time

    avg_training_time = total_training_time / num_trials
    avg_training_set_score = total_training_set_score / num_trials
    avg_training_prediction_time = total_training_prediction_time / num_trials
    avg_test_set_score = total_test_set_score / num_trials
    avg_test_prediction_time = total_test_prediction_time / num_trials

    if verbose:
        print "F1 score for training set: {}".format(avg_training_set_score)
    if verbose:
        print "F1 score for test set: {}".format(avg_test_set_score)

    return (avg_training_time, avg_training_set_score, avg_training_prediction_t
ime, avg_test_set_score, \
```

```
            avg_test_prediction_time)


# Run the helper function above for desired subsets of training data
# Note: Keep the test set constant
def train_predict_different_training_set_sizes(clf, training_set_sizes):
    training_times = []
    prediction_times = []
    training_set_scores = []
    test_set_scores = []
    for training_set_size in training_set_sizes:
        sample = np.random.choice(num_train, training_set_size, replace=False)
        sample_X_train = X_train.iloc[sample, :]
        sample_y_train = y_train.iloc[sample]
        training_time, training_set_score, training_prediction_time, test_set_sc
ore, test_prediction_time \
            = train_predict(clf, sample_X_train, sample_y_train, X_test, y_test,
False)
        training_times.append(training_time)
        prediction_times.append(training_prediction_time + test_prediction_time)
        training_set_scores.append(training_set_score)
        test_set_scores.append(test_set_score)

    return (training_times, prediction_times, training_set_scores, test_set_scor
es)

import list_table

def create_table(training_set_sizes, training_times, prediction_times, training_
set_scores, test_set_scores):
    # This method based on article by Caleb Madrigal
    #      http://calebmadrigal.com/display-list-as-table-in-ipython-notebook/
    table = list_table.ListTable()
    table.append([ \
                  'training set size', 'training time (secs)', 'prediction time
(secs)', 'F1 score on training set', \
                  'F1 score on test set'])
    for i in range(len(training_set_sizes)):
        format = "{:.3f}"
        table.append( \
                      [training_set_sizes[i], format.format(training_times[i]), f
ormat.format(prediction_times[i]), \
                       training_set_scores[i], test_set_scores[i]])
    return table

def analyze_classifier(clf):
    # Train a model
    # Fit model to training data
    _ = train_classifier(clf, X_train, y_train, True)  # note: using entire trai
ning set here
    #print clf  # you can inspect the learned model by printing it

    print "-----------------------------------------"
```

```
    # Predict on training set and compute F1 score

    train_f1_score = predict_labels(clf, X_train, y_train, True)[0]
    print "F1 score for training set: {}".format(train_f1_score)

    print "----------------------------------------"

    # Predict on test data
    print "F1 score for test set: {}".format(predict_labels(clf, X_test, y_test,
True)[0])

    print "----------------------------------------"

training_set_sizes = [100, 200, 300]
```

Decision Tree

- Decision trees are a type of supervised learning that can be used for classification problems.
  - Strengths
    - Easy to use since little data preparation is needed. For example, values do not need to be normalized nor do dummy variables need to be created for categorical variables.
    - Easy to interpret graphically
  - Weaknesses
    - Prone to overfitting
- A decision tree is a good model to try for this problem since decision tree outputs are humanly comprehensible and could identify specific causes of student failure.

In [8]:

```
# Choose a model, import it and instantiate an object
from sklearn.tree import DecisionTreeClassifier
dt_clf = DecisionTreeClassifier()
analyze_classifier(dt_clf)

# Train and predict using different training set sizes
training_times, prediction_times, training_set_scores, test_set_scores = \
    train_predict_different_training_set_sizes(dt_clf, training_set_sizes)
create_table(training_set_sizes, training_times, prediction_times, training_set_
scores, test_set_scores)
```

```
Training DecisionTreeClassifier...
Done!
Training time (secs): 0.003
------------------------------------------
Predicting labels using DecisionTreeClassifier...
Done!
Prediction time (secs): 0.000
F1 score for training set: 1.0
------------------------------------------
Predicting labels using DecisionTreeClassifier...
Done!
Prediction time (secs): 0.000
F1 score for test set: 0.719424460432
------------------------------------------
```

Out[8]:

training set size	training time (secs)	prediction time (secs)	F1 score on training set	F1 score on test set
100	0.001	0.000	1.0	0.616634120272
200	0.001	0.000	1.0	0.659815114576
300	0.002	0.000	1.0	0.732481456936

Support Vector Machine

- Support vector machines are a type of supervised learning that can be used for classification problems.
  - Strengths
    - Work well in complicated domains with a clear margin of separation
  - Weaknesses
    - Don't work well in large data sets because of cubic computation time
    - Don't work well in data sets with lots of noise
- I think support vector machines will work well with this data set since it is small and there should not be much noise (that is, all the data should be accurate).

In [9]:

```
# Choose a model, import it and instantiate an object
from sklearn import svm
svm_clf = svm.SVC()
analyze_classifier(svm_clf)

# Train and predict using different training set sizes
training_times, prediction_times, training_set_scores, test_set_scores = \
    train_predict_different_training_set_sizes(svm_clf, training_set_sizes)
create_table(training_set_sizes, training_times, prediction_times, training_set_
scores, test_set_scores)
```

```
Training SVC...
Done!
Training time (secs): 0.008
------------------------------------------
Predicting labels using SVC...
Done!
Prediction time (secs): 0.007
F1 score for training set: 0.871459694989
------------------------------------------
Predicting labels using SVC...
Done!
Prediction time (secs): 0.002
F1 score for test set: 0.776315789474
------------------------------------------
```

Out[9]:

training set size	training time (secs)	prediction time (secs)	F1 score on training set	F1 score on test set
100	0.001	0.002	0.8875	0.787096774194
200	0.003	0.004	0.85618729097	0.781456953642
300	0.006	0.006	0.871459694989	0.776315789474

Logistic Regression

- Logistic regression is a type of supervised learning that can be used for classification problems. Logistic regression is a regression model.
  - Strengths
    - Fast training
    - Linear model
  - Weaknesses (see The Disadvantages of Logistic Regression (http://www.ehow.com/info_8574447_disadvantages-logistic-regression.html) for details)
    - Independent observations required
    - Overfitting the model
- According to the Microsoft Azure Machine Learning: Algorithm Cheat Sheet (https://azure.microsoft.com/en-us/documentation/articles/machine-learning-algorithm-cheat-sheet/), two-class logistic regression is a good algorithm for two-class classification.

In [10]:

```
# Choose a model, import it and instantiate an object
from sklearn.linear_model import LogisticRegression
lr_clf = LogisticRegression()
analyze_classifier(lr_clf)

# Train and predict using different training set sizes
training_times, prediction_times, training_set_scores, test_set_scores = \
    train_predict_different_training_set_sizes(lr_clf, training_set_sizes)
create_table(training_set_sizes, training_times, prediction_times, training_set_
scores, test_set_scores)
```

```
Training LogisticRegression...
Done!
Training time (secs): 0.039
------------------------------------------
Predicting labels using LogisticRegression...
Done!
Prediction time (secs): 0.004
F1 score for training set: 0.857142857143
------------------------------------------
Predicting labels using LogisticRegression...
Done!
Prediction time (secs): 0.001
F1 score for test set: 0.695652173913
------------------------------------------

Out[10]:
```

training set size	training time (secs)	prediction time (secs)	F1 score on training set	F1 score on test set
100	0.001	0.000	0.909090909091	0.69696969697
200	0.001	0.000	0.852233676976	0.719424460432
300	0.003	0.000	0.857142857143	0.695652173913

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?
  - Let's compare the performance of all three candidate algorithms (on the test set):

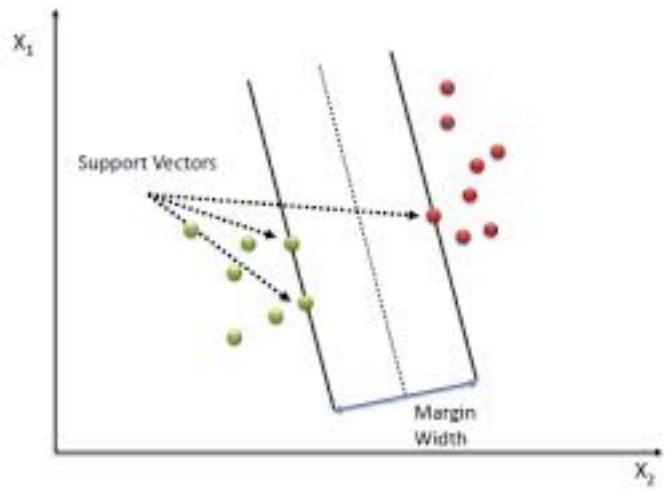
Algorithm	**F_1 score**	**Training time (secs)**	**Prediction time (secs)**
Decision Tree	0.719424460432	0.003	0.000
Support Vector Machine	0.776315789474	0.008	0.002
Logistic Regression	0.695652173913	0.039	0.001

  Support vector machines get the highest F_1 scores on the test set. Assuming the cost of using support vector machines is not prohibitively high, it seems to be the best choice. Training will only need to be done once initially and then probably at the end of each school year as we get more data on whether students pass or fail. Since training time is only 0.008 seconds, this is a small annual cost. The test set is 95 and the prediction time on the test set is 0.002 seconds. Assuming the student population is 1600 (which would be a large high school (https://nces.ed.gov/pubs2001/overview/table05.asp)), the time to make predictions on each member of the high school population is (0.002 seconds / 95 students) * 1600 students = 0.03 seconds. This could be run nightly without significant

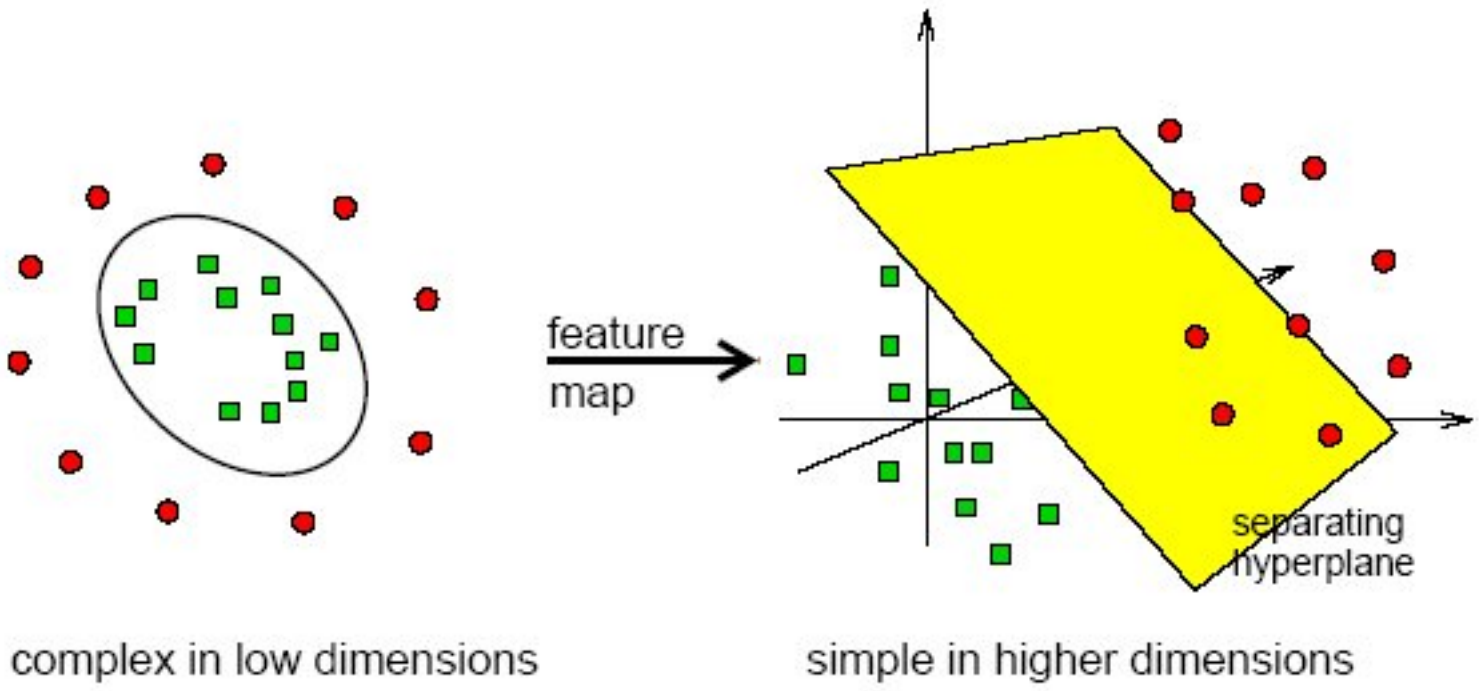
cost. So support vector machines is the way to go.

- 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).
    - A support vector machine is a machine learning algorithm that learns from data you give it on past students and whether they passed or failed. It does this by finding lines of separation in the feature space between students who pass and students who fail. It works to find large margins between these two groups. It can then make predictions on whether current students will pass or fail given data about them.

      A support vector machine is trained by trying to find a linear separator (or a planar, hyper-planar, and so on, separator in higher-dimensional spaces) between the positive test cases and the negative tests cases. It wants to maximize the *margin width* around this linear separator. A larger margin width leads to less overfitting. The points close to the margin (on both sides) are the *support vectors*. The support vectors define the model. The support vectors are a small part of the data set. After the model is constructed, there is no need to store the remaining points and hence the model is small.



In some cases, there may appear to be no linear separator. However, it may be possible to find a linear separator in a higher dimensional space. This is known as the *kernel trick*.



The algorithm makes predictions for a data point by simply seeing which side of the separator the data point falls on.

- 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?

In [17]:

```
# Fine-tune your model and report the best F1 score
default_gamma = 1.0 / n_features
param_grid = {'kernel': ('linear', 'rbf', 'sigmoid', 'poly'), \
              'C': [0.01, 0.1, 1, 10, 100], \
              'gamma': [default_gamma / 10, \
                        default_gamma, \
                        default_gamma * 10, default_gamma * 100]}

from sklearn.metrics import f1_score
from sklearn.metrics import make_scorer
from sklearn.grid_search import GridSearchCV

f1_scorer = make_scorer(f1_score, pos_label="yes")
grid_search = GridSearchCV(svm_clf, param_grid, scoring=f1_scorer)
grid_search.fit(X_train, y_train)

kernel = grid_search.best_params_['kernel']
C = grid_search.best_params_['C']
gamma = grid_search.best_params_['gamma']
svm_clf = svm_clf.set_params(kernel=kernel, C=C, gamma=gamma)
svm_clf.fit(X_train, y_train)

print "Model's final F1 score (for test set): {}".format(predict_labels(svm_clf,
X_test, y_test, False)[0])
```

```
Model's final F1 score (for test set): 0.78431372549
```