Machine Learning Engineer Nanodegree

Supervised Learning

Project 2: Building a Student Intervention System

Welcome to the second project of the Machine Learning Engineer Nanodegree! In this notebook, some template code has already been provided for you, and it will be your job to implement the additional functionality necessary to successfully complete this project. Sections that begin with **'Implementation'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section and the specifics of the implementation are marked in the code block with a `'TODO'` statement. Please be sure to read the instructions carefully!

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

Question 1 - Classification vs. Regression

Your goal for this project is to identify students who might need early intervention before they fail to graduate. Which type of supervised learning problem is this, classification or regression? Why?

Answer: Considering that I need to identify students that need early intervention, I am assuming that I will also identify the Students that do not need any intervention. Based on that I am considering this Supervised Learning as a Classication problem.

Exploring the Data

Run the code cell below to load necessary Python libraries and load the student data. Note that the last column from this dataset, `'passed'`, will be our target label (whether the student graduated or didn't graduate). All other columns are features about each student.

In [3]:
```
# Import libraries
import numpy as np
import pandas as pd
from time import time
from sklearn.metrics import f1_score

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

```
Student data read successfully!
```

Implementation: Data Exploration

Let's begin by investigating the dataset to determine how many students we have information on, and learn about the graduation rate among these students. In the code cell below, you will need to compute the following:

- The total number of students, `n_students`.
- The total number of features for each student, `n_features`.
- The number of those students who passed, `n_passed`.
- The number of those students who failed, `n_failed`.
- The graduation rate of the class, `grad_rate`, in percent (%).

In [4]:
```
# TODO: Calculate number of students
n_students = len(student_data)

# TODO: Calculate number of features
n_features = (len(student_data.columns) - 1)

# TODO: Calculate passing students
n_passed = sum(student_data['passed']=='yes')

# TODO: Calculate failing students
n_failed = sum(student_data['passed']=='no')

# TODO: Calculate graduation rate
grad_rate = (n_passed * 100.0) / n_students

# Print the results
print "Total number of students: {}".format(n_students)
print "Number of features: {}".format(n_features)
print "Number of students who passed: {}".format(n_passed)
print "Number of students who failed: {}".format(n_failed)
print "Graduation rate of the class: {:.2f}%".format(grad_rate)
```

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

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.

Run the code cell below to separate the student data into feature and target columns to see if any features are non-numeric.

In [5]:
```
# Extract feature columns
feature_cols = list(student_data.columns[:-1])

# Extract target column 'passed'
target_col = student_data.columns[-1]

# Show the list of columns
print "Feature columns:\n{}".format(feature_cols)
print "\nTarget column: {}".format(target_col)

# Separate the data into feature data and target data (X_all and y_all
, respectively)
X_all = student_data[feature_cols]
y_all = student_data[target_col]

# Show the feature information by printing the first five rows
print "\nFeature values:"
print X_all.head()
```

```
Feature columns:
['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()` (//htmlpreview.github.io/?https://github.com/robertodias/machine_learning/blob/master/python/student_intervention/student_interventic function to perform this transformation. Run the code cell below to perform the preprocessing routine discussed in this section.

In [6]:
```
def preprocess_features(X):
    ''' Preprocesses the student data and converts non-numeric binary variables into
        binary (0/1) variables. Converts categorical variables into dummy variables. '''

    # Initialize new output DataFrame
    output = pd.DataFrame(index = X.index)

    # Investigate each feature column for the data
    for col, col_data in X.iteritems():

        # If data type is non-numeric, replace all yes/no values with 1/0
        if col_data.dtype == object:
            col_data = col_data.replace(['yes', 'no'], [1, 0])

        # If data type is categorical, convert to dummy variables
        if col_data.dtype == object:
            # Example: 'school' => 'school_GP' and 'school_MS'
            col_data = pd.get_dummies(col_data, prefix = col)

        # Collect the revised columns
        output = output.join(col_data)

    return output

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

```
Processed feature columns (48 total features):
['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', 'Mjob_services', 'Mjob_teacher', 'Fjob_at_home', 'Fjob_health', 'Fjob_other', 'Fjob_services', 'Fjob_teacher', 'reason_course', 'reason_home', 'reason_other', 'reason_reputation', 'guardian_father', 'guardian_mother', 'guardian_other', 'traveltime', 'studytime', 'failures', 'schoolsup', 'famsup', 'paid', 'activities', 'nursery', 'higher', 'internet', 'romantic', 'famrel', 'freetime', 'goout', 'Dalc', 'Walc', 'health', 'absences']
```

Implementation: Training and Testing Data Split

So far, we have converted all *categorical* features into numeric values. For the next step, we split the data (both features and corresponding labels) into training and test sets. In the following code cell below, you will need to implement the following:

- Randomly shuffle and split the data (`X_all`, `y_all`) into training and testing subsets.
  - Use 300 training points (approximately 75%) and 95 testing points (approximately 25%).
  - Set a `random_state` for the function(s) you use, if provided.
  - Store the results in `X_train`, `X_test`, `y_train`, and `y_test`.

In [21]:
```
# TODO: Import any additional functionality you may need here
from sklearn.cross_validation import train_test_split

# TODO: Set the number of training points
num_train = 300

# Set the number of testing points
num_test = X_all.shape[0] - num_train

# TODO: Shuffle and split the dataset into the number of training and testing points above
X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=num_test, random_state=33, stratify=y_all)

# Show the results of the split
print "Training set has {} samples.".format(X_train.shape[0])
print "Testing set has {} samples.".format(X_test.shape[0])
```

```
Training set has 300 samples.
Testing set has 95 samples.
```

Training and Evaluating Models

In this section, you will choose 3 supervised learning models that are appropriate for this problem and available in `scikit-learn`. You will first discuss the reasoning behind choosing these three models by considering what you know about the data and each model's strengths and weaknesses. You will then fit the model to varying sizes of training data (100 data points, 200 data points, and 300 data points) and measure the F_1 score. You will need to produce three tables (one for each model) that shows the training set size, training time, prediction time, F_1 score on the training set, and F_1 score on the testing set.

The following supervised learning models are currently available in `scikit-learn` (http://scikit-learn.org/stable/supervised_learning.html) that you may choose from:

- Gaussian Naive Bayes (GaussianNB)
- Decision Trees
- Ensemble Methods (Bagging, AdaBoost, Random Forest, Gradient Boosting)
- K-Nearest Neighbors (KNeighbors)
- Stochastic Gradient Descent (SGDC)
- Support Vector Machines (SVM)
- Logistic Regression

Question 2 - Model Application

List three supervised learning models that are appropriate for this problem. For each model chosen

- Describe one real-world application in industry where the model can be applied. *(You may need to do a small bit of research for this — give references!)*
- What are the strengths of the model; when does it perform well?
- What are the weaknesses of the model; when does it perform poorly?
- What makes this model a good candidate for the problem, given what you know about the data?

Answer:

Gaussian Naive Bayes

Today, Naive Bayes is one of the most used Supervised Learning models. A very important example is related with Cancer treatment, "Application of the Naïve Bayesian Classifier to optimize treatment decisions" Book (Radiotherapy and Oncology Volume 86, Issue 2, February 2008, Pages 211–216). Another example is related with consumer credit scoring. One real example is with ZOPA a loan company (https://www.zopa.com (https://www.zopa.com)) that uses the NB to classify and define a credit score for client (http://ieeexplore.ieee.org/document/6282321/?reload=true (http://ieeexplore.ieee.org/document/6282321/?reload=true)). Another important application of NB is related with the email classifier applied to identify SPAMs, it uses a method called bag of words to quantify the words and provide a dataset for the Naive Bayes application (https://en.wikipedia.org/wiki/Naive_Bayes_spam_filtering (https://en.wikipedia.org/wiki/Naive_Bayes_spam_filtering)).

- **Strengths** It is considered very easy to implement and efficient for training, in terms of the total number of computations needed. Also, naive Bayes performs well on many different training tasks. The "good" part of being naive is that even with a small number of training data it can satisfactory estimate the parameters necessary for classification
- **Weaknesses** Because of the strong conditional independence assumption placed on the attributes in the model, there are situations where naive Bayes is not appropriate. The "naive" part of the algorithm can potencially ends with bad predictions, in the case where two ore more features, together, have a completely different meaning than when are all features being analysed independently. i.e.: "Chicago Bulls" means the Basketball team but when seeing in the naive way are just Chicago (city) and bulls (the animal).
- **Why a good candidate** Considering our problem a Classification problem, Naive Bayes is one of the most simple and effective Classifiers models, even if the expected assumption doesn't hold, a Naive Bayes classifier still often does a great job in practice.

Decision Trees

Astronomy: Use of decision trees for filtering noise from Hubble Space Telescope images. Decision trees have helped in star-galaxy classification, determining galaxy counts and discovering quasars in the Second Palomar Sky Survey (http://www.cbcb.umd.edu/~salzberg/docs/murthy_thesis/survey/node32.html (http://www.cbcb.umd.edu/~salzberg/docs/murthy_thesis/survey/node32.html)). Also it was recently successfuly applied to NLP part-of-speech disambiguation and unknown word guessing (Decision Trees and NLP: A Case Study in POS Tagging, https://pdfs.semanticscholar.org/f346/e045535f6928c4c404b86d2d6b317c3b5d8b.pdf (https://pdfs.semanticscholar.org/f346/e045535f6928c4c404b86d2d6b317c3b5d8b.pdf)).

- **Strengths** Decision trees are simple to understand and interpret. The induction of decision trees is a method that generates approximations to discrete-valued functions and has been shown, experimentally, to provide robust performance in the presence of noise.
- **Weaknesses** Based on the concept of the Decision tree and the way the nodes are associate with each others, there is a considerable risk of overfitting. The decision tree can change significantly at any dataset change. This is not desirable as we want our classification algorithm to be pretty robust to noise and be able to generalize well to future observed data. Considering the growth of the tree,

for each new feature the algorithm may need to recalculate the entire tree resulting in bad performance (Utgoff, P. E. (1989). Incremental induction of decision trees. Machine learning, 4(2), 161-186.).
- **Why a good candidate** Considering our problem a Classification problem, Decision trees have long been considered as one of the most practical and straightforward approaches to classification (Quinlan, J.R. (1986) Induction of Decision Trees, Machine Learning, 1:81-106.).

Logistic Regression An example of Logistic Regression might be to predict whether an American voter will vote Democratic or Republican, based on age, income, sex, race, state of residence, votes in previous elections (Harrell, Frank E. (2001). Regression Modeling Strategies. Springer-Verlag. ISBN 0-387-95232-2.). It is also used in marketing applications such as prediction of a customer's propensity to purchase a product or halt a subscription (Berry, Michael J.A (1997). Data Mining Techniques For Marketing, Sales and Customer Support. Wiley. p. 10.) web: http://en.wikipedia.org/wiki/Logistic_regression (http://en.wikipedia.org/wiki/Logistic_regression).

- **Strengths** In Logistic Regression you don't have to worry if your features are correlated, like you do in Naive Bayes. You also easily update your model to take in new data and have a nice probabilistic interpretation, unlike decision trees. (http://blog.echen.me/2011/04/27/choosing-a-machine-learning-classifier/ (http://blog.echen.me/2011/04/27/choosing-a-machine-learning-classifier/))
- **Weaknesses** Logistic regression attempts to predict outcomes based on a set of independent variables, but those models are vulnerable to overfit. That is, the models can appear to have more predictive power than they actually do as a result of sampling bias. The features need to be well selected, if we include the wrong independent variables, the model will have little to no predictive value. (http://classroom.synonym.com/disadvantages-logistic-regression-8574447.html (http://classroom.synonym.com/disadvantages-logistic-regression-8574447.html))
- **Why a good candidate** Considering our problem a Classification problem, Logistic regression is a pretty well-behaved classification algorithm that can be trained as long as you expect your features to be roughly linear and the problem to be linearly separable. It is also pretty robust to noise and you can avoid overfitting when applying L1 or L2 regularization (https://web.stanford.edu/~boyd/papers/admm/logreg-l1/logreg.html (https://web.stanford.edu/~boyd/papers/admm/logreg-l1/logreg.html)).

P.S.: Following the Machine Learning Map (http://scikit-learn.org/stable/tutorial/machine_learning_map/ (http://scikit-learn.org/stable/tutorial/machine_learning_map/)) It suggest us to use **Linear Support Vector Machines** or **Naive Bayes to solve this problem**. I will keep this map close to me on my next adventures :)

Setup

Run the code cell below to initialize three helper functions which you can use for training and testing the three supervised learning models you've chosen above. The functions are as follows:

- `train_classifier` - takes as input a classifier and training data and fits the classifier to the data.
- `predict_labels` - takes as input a fit classifier, features, and a target labeling and makes predictions using the F_1 score.
- `train_predict` - takes as input a classifier, and the training and testing data, and performs `train_clasifier` and `predict_labels`.
    - This function will report the F_1 score for both the training and testing data separately.

In [22]:
```
def train_classifier(clf, X_train, y_train):
    ''' Fits a classifier to the training data. '''

    # Start the clock, train the classifier, then stop the clock
    start = time()
    clf.fit(X_train, y_train)
    end = time()

    # Print the results
    print "Trained model in {:.4f} seconds".format(end - start)


def predict_labels(clf, features, target):
    ''' Makes predictions using a fit classifier based on F1 score. ''
'

    # Start the clock, make predictions, then stop the clock
    start = time()
    y_pred = clf.predict(features)
    end = time()

    # Print and return results
    print "Made predictions in {:.4f} seconds.".format(end - start)
    return f1_score(target.values, y_pred, pos_label='yes')


def train_predict(clf, X_train, y_train, X_test, y_test):
    ''' Train and predict using a classifer based on F1 score. '''

    # Indicate the classifier and the training set size
    print "Training a {} using a training set size of {}. . .".format(
clf.__class__.__name__, len(X_train))

    # Train the classifier
    train_classifier(clf, X_train, y_train)

    # Print the results of prediction for both training and testing
    print "F1 score for training set: {:.4f}.".format(predict_labels(c
lf, X_train, y_train))
    print "F1 score for test set: {:.4f}.".format(predict_labels(clf,
X_test, y_test))
```

Implementation: Model Performance Metrics

With the predefined functions above, you will now import the three supervised learning models of your choice and run the `train_predict` function for each one. Remember that you will need to train and predict on each classifier for three different training set sizes: 100, 200, and 300. Hence, you should expect to have 9 different outputs below — 3 for each model using the varying training set sizes. In the following code cell, you will need to implement the following:

- Import the three supervised learning models you've discussed in the previous section.
- Initialize the three models and store them in `clf_A`, `clf_B`, and `clf_C`.
  - Use a `random_state` for each model you use, if provided.
  - **Note:** Use the default settings for each model — you will tune one specific model in a later section.
- Create the different training set sizes to be used to train each model.
  - *Do not reshuffle and resplit the data! The new training points should be drawn from `X_train` and `y_train`.*
- Fit each model with each training set size and make predictions on the test set (9 in total).
  Note: Three tables are provided after the following code cell which can be used to store your results.

In [31]:
```
# TODO: Import the three supervised learning models from sklearn
# from sklearn import model_A
from sklearn.naive_bayes import GaussianNB

# from sklearn import model_B
from sklearn import tree

# from skearln import model_C
from sklearn.linear_model import LogisticRegression

# TODO: Initialize the three models
clf_A = GaussianNB()
clf_B = tree.DecisionTreeClassifier(random_state=33)
clf_C = LogisticRegression(C=1e5, random_state=33)

# TODO: Set up the training set sizes
X_train_100, X_test_100, y_train_100, y_test_100 = train_test_split(X_
all,y_all, test_size=100, random_state=33)
X_train_200, X_test_200, y_train_200, y_test_200 = train_test_split(X_
all,y_all, test_size=200, random_state=33)
X_train_300, X_test_300, y_train_300, y_test_300 = train_test_split(X_
all,y_all, test_size=300, random_state=33)

# TODO: Execute the 'train_predict' function for each classifier and e
ach training set size
for clf in [clf_A, clf_B, clf_C]:
    for n_train in [100, 200, 300]:
        train_predict(clf, X_train[:n_train], y_train[:n_train], X_tes
t, y_test)
```

```
Training a GaussianNB using a training set size of 100. . .
Trained model in 0.0032 seconds
Made predictions in 0.0008 seconds.
F1 score for training set: 0.3333.
Made predictions in 0.0013 seconds.
F1 score for test set: 0.4235.
Training a GaussianNB using a training set size of 200. . .
Trained model in 0.0021 seconds
Made predictions in 0.0014 seconds.
F1 score for training set: 0.7865.
Made predictions in 0.0005 seconds.
F1 score for test set: 0.7857.
Training a GaussianNB using a training set size of 300. . .
Trained model in 0.0021 seconds
Made predictions in 0.0014 seconds.
F1 score for training set: 0.7855.
Made predictions in 0.0006 seconds.
F1 score for test set: 0.7737.
```

```
Training a DecisionTreeClassifier using a training set size of 100.
. .
Trained model in 0.0013 seconds
Made predictions in 0.0002 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0003 seconds.
F1 score for test set: 0.7132.
Training a DecisionTreeClassifier using a training set size of 200.
. .
Trained model in 0.0014 seconds
Made predictions in 0.0002 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0002 seconds.
F1 score for test set: 0.7206.
Training a DecisionTreeClassifier using a training set size of 300.
. .
Trained model in 0.0019 seconds
Made predictions in 0.0002 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0001 seconds.
F1 score for test set: 0.7424.
Training a LogisticRegression using a training set size of 100. . .
Trained model in 0.0026 seconds
Made predictions in 0.0001 seconds.
F1 score for training set: 0.8936.
Made predictions in 0.0001 seconds.
F1 score for test set: 0.6718.
Training a LogisticRegression using a training set size of 200. . .
Trained model in 0.0020 seconds
Made predictions in 0.0002 seconds.
F1 score for training set: 0.8483.
Made predictions in 0.0001 seconds.
F1 score for test set: 0.8163.
Training a LogisticRegression using a training set size of 300. . .
Trained model in 0.0031 seconds
Made predictions in 0.0002 seconds.
F1 score for training set: 0.8322.
Made predictions in 0.0002 seconds.
F1 score for test set: 0.7971.
```

Tabular Results

Edit the cell below to see how a table can be designed in Markdown (//htmlpreview.github.io/?https://github.com/robertodias/machine_learning/blob/master/python/student_intervention/student_interventic
You can record your results from above in the tables provided.

Classifer 1 - Gaussian Naive Bayes

Training Set Size	Training Time	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	0.0036	0.0009	0.3333	0.4235
200	0.0034	0.0014	0.7865	0.7857
300	0.0018	0.0010	0.7855	0.7737

Classifer 2 - Decision Tree

Training Set Size	Training Time	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	0.0010	0.0003	1.0000	0.7132
200	0.0019	0.0002	1.0000	0.7206
300	0.0020	0.0002	1.0000	0.7424

Classifer 3 - Logistic Regression

Training Set Size	Training Time	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	0.0030	0.0001	0.8936	0.6718
200	0.0028	0.0002	0.8483	0.8163
300	0.0033	0.0002	0.8322	0.7971

Choosing the Best Model

In this final section, you will choose from the three supervised learning models the *best* model to use on the student data. You will then perform a grid search optimization for the model over the entire training set (`X_train` and `y_train`) by tuning at least one parameter to improve upon the untuned model's F_1 score.

Question 3 - Choosing the Best Model

Based on the experiments you performed earlier, in one to two 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?

Answer:

The reason for using Logistic Regression for this problem is that the dependent variable passed represented by "1" and "0" are not cardinal numbers. If the problem were changed so that passed was replaced with the grade 0–100 (cardinal numbers), then simple regression analysis could be used.

Walking thru the results of the others models, I could see that Naive Bayes presented the best times in learning, and lower CPU costs but lost its prediction scores when I increased the training set size. Same for Decision Tree that showed a high predictive score on training but a very poor prediction score in the real data, what to me looked very like it was orverfitting.

On the other hand, the Logistic Regression showed promissing results, despite is training time showed to be bigger than the other models it kept the prediction score high in real data. Another important point, that make me choose from Naive Bayes and Logistic Regression was that fact that Logistic Regression can handle correlated features way better than Naive Bayes, what I consider important in our scenario.

Question 4 - Model in Layman's Terms

In one to two paragraphs, explain to the board of directors in layman's terms how the final model chosen is supposed to work. Be sure that you are describing the major qualities of the model, such as how the model is trained and how the model makes a prediction. Avoid using advanced mathematical or technical jargon, such as describing equations or discussing the algorithm implementation.

Answer: Logistic Regression is a regression model that makes predictions based on probabilities. It's focused in the conditional probability distribution of Y given X.

The whole idea is to, based on the data you have, find a generic function that could be applied to any new data entry to produce a predictions. Logistic Regression works as a binary (0 and 1's) logistic model, which means it is used to estimate the probability of binary responses considering multiple entry data features.

Logistic Regression is a method applied to find a binary output of Y, considering a linearly combined input of X's. Binomial or binary logistic regression deals with situations in which the observed outcome for a dependent variable can have only two possible types, "0" and "1" and the logistic function (sigmoid) is used at the core of this model to guarantee the binary behavior the results/predictions.

So, in our scenario, the traning data help us to create a equation that when applied to those features will produce a binary result that best matches with the training result we have. After the "fit" of our model we can then use it to generalize predictions using real data. In our case we are going to enter new students data and then our trained model will answers back if that student will pass or fail the graduation.

Implementation: Model Tuning

Fine tune the chosen model. Use grid search (`GridSearchCV`) with at least one important parameter tuned with at least 3 different values. You will need to use the entire training set for this. In the code cell below, you will need to implement the following:

- Import `sklearn.grid_search.gridSearchCV` (http://scikit-learn.org/stable/modules/generated/sklearn.grid_search.GridSearchCV.html) and `sklearn.metrics.make_scorer` (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html).
- Create a dictionary of parameters you wish to tune for the chosen model.
  - Example: `parameters = {'parameter' : [list of values]}`.
- Initialize the classifier you've chosen and store it in `clf`.
- Create the F_1 scoring function using `make_scorer` and store it in `f1_scorer`.
  - Set the `pos_label` parameter to the correct value!
- Perform grid search on the classifier `clf` using `f1_scorer` as the scoring method, and store it in `grid_obj`.
- Fit the grid search object to the training data (`X_train`, `y_train`), and store it in `grid_obj`.

In [37]:
```
# TODO: Import 'GridSearchCV' and 'make_scorer'
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import fbeta_score, make_scorer

# TODO: Create the parameters list you wish to tune
parameters = [
    {'C': [1, 10, 100, 1000],
    'class_weight': ['balanced', None]}
]

# TODO: Initialize the classifier
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(C=1e5, random_state=33)

# TODO: Make an f1 scoring function using 'make_scorer'
f1_scorer = make_scorer(f1_score, pos_label="yes")

# TODO: Perform grid search on the classifier using the f1_scorer as t
he scoring method
grid_obj = GridSearchCV(clf, param_grid=parameters, scoring=f1_scorer,
cv=5, verbose=0)

# TODO: Fit the grid search object to the training data and find the o
ptimal parameters
grid_obj.fit(X_train, y_train)

# Get the estimator
clf = grid_obj.best_estimator_

# Report the final F1 score for training and testing after parameter t
uning
print "Tuned model has a training F1 score of {:.4f}.".format(predict_
labels(clf, X_train, y_train))
print "Tuned model has a testing F1 score of {:.4f}.".format(predict_l
abels(clf, X_test, y_test))
```

```
Made predictions in 0.0002 seconds.
Tuned model has a training F1 score of 0.8349.
Made predictions in 0.0001 seconds.
Tuned model has a testing F1 score of 0.8085.
```

Question 5 - Final F_1 Score

What is the final model's F_1 score for training and testing? How does that score compare to the untuned model?

Answer: The final model's F1 results were 0.8349 for training and 0.8085 for testing. Both traning and testing were slightly better than the untuned model, untuned Training was 0.83222 and untuned testing was 0.7971.

Tunning here looks to be a good option, even if we are still not reaching high confidence predictions.

> **Note**: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.