Machine Learning Engineer Nanodegree

Supervised Learning

Project: 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: Classification.

Regression is used to predict continuous values. Classification is used to predict which class a data point is in(discrete value). In this problem, we need to predict whether a student is going to pass or fail (with likelihood). So it should be classification 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 [2]:
```
# 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!
```

In [3]:
```
student_data.head()
```

Out[3]:

	school	sex	age	address	famsize	Pstatus	Medu	Fedu	Mjob	Fjob	...	interr
0	GP	F	18	U	GT3	A	4	4	at_home	teacher	...	no
1	GP	F	17	U	GT3	T	1	1	at_home	other	...	yes
2	GP	F	15	U	LE3	T	1	1	at_home	other	...	yes
3	GP	F	15	U	GT3	T	4	2	health	services	...	yes
4	GP	F	16	U	GT3	T	3	3	other	other	...	no

5 rows × 31 columns

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]:
```
student_data.columns[:]
```

Out[4]:
```
Index([u'school', u'sex', u'age', u'address', u'famsize', u'Pstatus', u'Med
u',
       u'Fedu', u'Mjob', u'Fjob', u'reason', u'guardian', u'traveltime',
       u'studytime', u'failures', u'schoolsup', u'famsup', u'paid',
       u'activities', u'nursery', u'higher', u'internet', u'romantic',
       u'famrel', u'freetime', u'goout', u'Dalc', u'Walc', u'health',
       u'absences', u'passed'],
      dtype='object')
```

In [5]:
```
# 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 = len(student_data[student_data['passed']=='yes'])

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

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

# 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 [6]:
```
# 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, respec
tively)
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', 'Fedu', 'Mj
ob', 'Fjob', 'reason', 'guardian', 'traveltime', 'studytime', 'failures', 'sc
hoolsup', 'famsup', 'paid', 'activities', 'nursery', 'higher', 'internet', 'r
omantic', 'famrel', 'freetime', 'goout', 'Dalc', 'Walc', 'health', 'absence
s']

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]
```

In [7]:
```
y_all.head()
```

Out[7]:
```
0     no
1     no
2    yes
3    yes
4    yes
Name: passed, dtype: object
```

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. Run the code cell below to perform the preprocessing routine discussed in this section.

In [8]:
```
def preprocess_features(X):
    ''' Preprocesses the student data and converts non-numeric binary variable
s into
        binary (0/1) variables. Converts categorical variables into dummy vari
ables. '''

    # 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.co
lumns), 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', 'Fjo
b_at_home', 'Fjob_health', 'Fjob_other', 'Fjob_services', 'Fjob_teacher', 're
ason_course', 'reason_home', 'reason_other', 'reason_reputation', 'guardian_f
ather', 'guardian_mother', 'guardian_other', 'traveltime', 'studytime', 'fail
ures', 'schoolsup', 'famsup', 'paid', 'activities', 'nursery', 'higher', 'int
ernet', 'romantic', 'famrel', 'freetime', 'goout', 'Dalc', 'Walc', 'health',
 'absences']
```

In [9]:
```
X_all.head()
```

Out[9]:

	school_GP	school_MS	sex_F	sex_M	age	address_R	address_U	famsize_GT3	fa
0	1.0	0.0	1.0	0.0	18	0.0	1.0	1.0	0.(
1	1.0	0.0	1.0	0.0	17	0.0	1.0	1.0	0.(
2	1.0	0.0	1.0	0.0	15	0.0	1.0	0.0	1.(
3	1.0	0.0	1.0	0.0	15	0.0	1.0	1.0	0.(
4	1.0	0.0	1.0	0.0	16	0.0	1.0	1.0	0.(

5 rows × 48 columns

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 [14]:
```
# 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,train_size=num
_train,random_state=1)

# 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.
```

In [15]:
```
X_train.shape
X_test.shape
```

Out[15]: (95, 48)

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: The models I chose are **GaussianNB, SVM, Descision Trees**

- GaussianNB
  - Strength: Very fast. Require small amount of data to train. Not much parameters to tune.
  - Weakness: 1. Requires the features to be independent. 2. The calculated probability could be 0, if the feature is not in the training data. Use laplace smoothing in this case. 3. For continuous features. It is common to use a binning procedure to make them discrete, but if you are not careful you can throw away a lot of information.
  - Example for Gaussian Naive Bayes applied in Healthcare: (http://www.ijcst.org/Volume3/Issue1/p19_3_1.pdf (http://www.ijcst.org/Volume3/Issue1/p19_3_1.pdf)); (http://www.ijcaonline.org/archives/volume148/number6/gayathri-2016-ijca-911146.pdf (http://www.ijcaonline.org/archives/volume148/number6/gayathri-2016-ijca-911146.pdf)) to classify breast cancer. Naive Bayes method is also widely used for spam email detection. But from the online examples from Udacity class (Intro to AI by Sebastian Thrun), if we just count the words, it should be multinomial Naive Bayes.
- SVM
  - Strength: 1. it has a regularisation parameter, which makes the user think about avoiding over-fitting. 2. it uses the kernel trick, so you can build in expert knowledge about the problem via engineering the kernel. 3. an SVM is defined by a convex optimisation problem (no local minima) for which there are efficient methods (e.g. SMO) 4. Compared with logistic regression, the hyper-plane of SVM is theoretically maximizing the "margin", thus less prone to overfitting.
  - Weakness: 1. relatively slow. 2. many hyper-parameters to tune. 3. require knowledge about which kernal to use.
  - SVM also has lots of applications. In the document (http://www3.ntu.edu.sg/home/elpwang/pdf_web/05_svm_basic.pdf (http://www3.ntu.edu.sg/home/elpwang/pdf_web/05_svm_basic.pdf)), from page 297 to the end, the authors talked about its applications in signal processing, Cancer diagnosis, Gas sensing, etc.
- Decision Tree
  - Strength: 1. Decision trees implicitly perform variable screening or feature selection; 2. Decision trees require relatively little effort from users for data preparation. 3. Nonlinear relationships between parameters do not affect tree performance.
  - Weakness: 1. easy to overfit. 2. have problems out-of-sample prediction. 3. problem of multicollinearity: when two variables both explain the same thing, a decision tree will greedily choose the best one, whereas many other methods will use them both.
  - Tons of applications using decision tree as well. (http://www.cbcb.umd.edu/~salzberg/docs/murthy_thesis/survey/node32.html (http://www.cbcb.umd.edu/~salzberg/docs/murthy_thesis/survey/node32.html)) shows that decision tree method can be used in Agriculture, Astronomy, Biomedical Engineering, Control System, ...

For this problem, the training set has 300 instances, each with 48 features. The dataset is not very large and no need for online processing. It is a binary classification problem (graduate or not). There is no reason the data is linearly seperable. According to all these reasons, I chose the **GaussianNB, SVM, Descision Trees** methods. However, these three methods are not the only suitable method for this problem.

Extra: The best way to compare these different classifers is to test these methods on the same datasets and look into the source code (or write simple source code for each one). I know some of them very clearly (method and how to write source code). But for the others (e.g. random forest), I just know the principle idea and haven't written sample code by myself. Together with the information I collected online, I summarize the main differences between these classifiers in the followings:

Different classifiers may perform differently for different problems. To choose the right classifier, we need to think about:

1. The size of the data. Is efficiency a big concern?
2. Is data linearly seperable?
3. Is high variance a bigger concern or high bias?
4. Require probabilities for estimation?
5. ...

- Naive Bayes
    1. Fast and easy to build
    2. Assume the features are independent of each other ("Naive").
    3. Can assume various distribution for the features, e.g. Gaussian, Multinomial, Bernoulli.
    4. Works naturally well with multi-class data.
    5. Does not require the data to be linearly seperable.
    6. Provide probabilities for estimation.
- Decision Trees
    1. Fast and easy to build
    2. No assumption for linear seperable or independence.
    3. Likely to overfit.
    4. Does not provide probabilities for estimation.
        - **There a better way to implement Tree methods:**
        - Bagging: Reduce the variance. Generate data subsets by boost-strap resampling (sampling with replacement). Then take average of all the results. (Random forest is a fancy bagging.)
        - Boosting: Get better prediction, reduce the bias. Give mis-classified sample higher weights to consider in each iteration. (Adaboosting, Gradient boosting)
- KNN
    1. Easy to understand.
    2. Take a lot of memory to store all the instances.
    3. No assumption for linear seperable or independence.
    4. If the dimension is high (lots of features), calculating the distance between data will be slow.
    5. Depending on the number of N to choose, the problem could be high-variance (small N) or high-bias (large N).
    6. Does not provide probabilities for estimation.
- SVM
    1. not easy to tune (which kernal, what hyper-parameter). However, this can be an advantage as flexibility.
    2. Memory efficent (only use part of the training data, support vectors)
    3. Bassically solve a linear programming problem (least-square minimization plus extra constraints).

4. The threshold is drawn to be in the middle of the edge of different classes (support vectors), thus is less prone to overfitting.
5. Fundamentally a binary linear classifier. However, implementing different kernels can be used for multi-class non-linear classifier.
6. Does not provide probabilities for estimation.

- Logistic Regression
  1. relatively easy to implement. Need to tune the regularization hyper-parameter.
  2. Bassically solve a convex minimization problem.
  3. The threshold is drawn to seperate the data using a hyper-plane. Unlike SVM, this threshold does not need to be in the middle of the "support vectors" (edge of data with different classes). Not as good as SVM in preventing overfitting.
  4. Fundamentally a binary linear classifier. It can deal with multi-class data too. But cannot deal with nonlinear problem.
  5. Provide probabilities for estimation.
     - Neural Network is just a complicated way to combine many layers of logistic regression.
- Stochastic Gradient Descent
  1. **SGD is not a different classifer.** SGD is just a different way to perform gradient descent, compared with steepest gradient descent method.
  2. SGD can use either SVM or logitstic regression to design the loss function (check sklearn manual).
  3. Compared with steepest descent or batch gradient descent method, SGD is faster (each iteration does not require all the data to calculate the perfect gradient) and is an online method.
  4. Better has lots of data. For small dataset, just use steepest descent method.

KNN, SVM, Logistic regression should all be sensitive to feature scaling (thus should SGD)

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 [16]:
```
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.__cl
ass__.__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(clf, X_tr
ain, 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 [18]:
```
# TODO: Import the three supervised learning models from sklearn
# from sklearn import model_A
# from sklearn import model_B
# from skearln import model_C
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier

# TODO: Initialize the three models
clf_A = GaussianNB()
clf_B = SVC(kernel='rbf',random_state=1)
clf_C = DecisionTreeClassifier(random_state=1)

# TODO: Set up the training set sizes
X_train_100, _, y_train_100, _ = train_test_split(X_train,y_train,train_size=1
00,random_state=1)

X_train_200, _, y_train_200, _ = train_test_split(X_train,y_train,train_size=2
00,random_state=1)

X_train_300=X_train
y_train_300=y_train

# TODO: Execute the 'train_predict' function for each classifier and each trai
ning set size
# train_predict(clf, X_train, y_train, X_test, y_test)

for clf in [clf_A, clf_B, clf_C]:
    for Xtrain, ytrain in zip([X_train_100, X_train_200, X_train_300],[y_train
_100, y_train_200, y_train_300]):
        print 20*'*'
        train_predict(clf, Xtrain, ytrain, X_test, y_test)
    print 30*'-'
```

```
**********
Training a GaussianNB using a training set size of 100. . .
Trained model in 0.0010 seconds
Made predictions in 0.0000 seconds.
F1 score for training set: 0.7917.
Made predictions in 0.0000 seconds.
F1 score for test set: 0.7606.
**********
Training a GaussianNB using a training set size of 200. . .
Trained model in 0.0010 seconds
Made predictions in 0.0000 seconds.
F1 score for training set: 0.7765.
Made predictions in 0.0000 seconds.
F1 score for test set: 0.6875.
**********
Training a GaussianNB using a training set size of 300. . .
Trained model in 0.0010 seconds
Made predictions in 0.0010 seconds.
F1 score for training set: 0.7921.
Made predictions in 0.0010 seconds.
F1 score for test set: 0.6720.
------------------------------
**********
Training a SVC using a training set size of 100. . .
Trained model in 0.0010 seconds
Made predictions in 0.0010 seconds.
F1 score for training set: 0.8689.
Made predictions in 0.0000 seconds.
F1 score for test set: 0.7846.
**********
Training a SVC using a training set size of 200. . .
Trained model in 0.0040 seconds
Made predictions in 0.0020 seconds.
F1 score for training set: 0.8581.
Made predictions in 0.0010 seconds.
F1 score for test set: 0.8571.
**********
Training a SVC using a training set size of 300. . .
Trained model in 0.0060 seconds
Made predictions in 0.0040 seconds.
F1 score for training set: 0.8584.
Made predictions in 0.0010 seconds.
F1 score for test set: 0.8462.
------------------------------
**********
Training a DecisionTreeClassifier using a training set size of 100. . .
Trained model in 0.0010 seconds
Made predictions in 0.0000 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0010 seconds.
F1 score for test set: 0.6612.
**********
Training a DecisionTreeClassifier using a training set size of 200. . .
Trained model in 0.0010 seconds
Made predictions in 0.0010 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0000 seconds.
```

```
F1 score for test set: 0.7463.
**********
Training a DecisionTreeClassifier using a training set size of 300. . .
Trained model in 0.0020 seconds
Made predictions in 0.0000 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0000 seconds.
F1 score for test set: 0.6984.
------------------------------
```

Tabular Results

Edit the cell below to see how a table can be designed in Markdown (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet#tables). 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	0.001	0.000	0.7917	0.7606
200	0.001	0.000	0.7765	0.6875
300	0.001	0.001	0.7921	0.6720

Classifer 2 - SVC with 'rbf' (radial basis function) kernal

Training Set Size	Training Time	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	0.001	0.000	0.8689	0.7846
200	0.004	0.001	0.8581	0.8571
300	0.006	0.001	0.8584	0.8462

Classifer 3 - Decision tree

Training Set Size	Training Time	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	0.001	0.001	1	0.6612
200	0.001	0.000	1	0.7463
300	0.002	0.000	1	0.6984

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:

Among the three tested models, I will suggest choosing the SVC model, for the following reasons:

- Training time. The training time using SVC is significantly longer than the other two models, and increases more quickly as the training size increases. The GaussianNB and Decision Tree methods are very fast. However, our training set is small so that overall the training time is all short.
- With 100, 200, 300 training sets, SVC gives the best F1 scores. And this score may even increase as the hyper-parameters are tuned. For GaussianNB method, there is no other parameter to tune (maybe the laplace smoothing factor), which means the score will not improve. And interestingly, the F1 for testing score decrease as the training size increase. For Decision tree method, The F1 score for testing is not very stable and for training is 1. This is very likely caused by overfitting. But we can also tune some parameters in decision tree to improve the prediction, e.g. the maximum tree depth.

So with our dataset, I would recommend SVC. If the dataset is too big, SVC may be too slow to run. In that case, decision tree could be the best choice regarding both computing time and score. Still, we should be careful with the hyper-parameters in decision tree method to avoid overfitting. And other ensemble method based on tree method could be even better, e.g. random forest.

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:

When we use SVC method to do classification, first, it use a big chunck of data to train a model, which is to find what combination of features and their values (in our case, 40 features, and some have discrete values, some have continuous values) is mostly likely to be related to their data labels (in our case, graduate or not). Since our data label is binary, the SVC method is just trying to make two groups of combinations of features and their values. One group will be the students who graduate, the other is the students failed to graduate. The graduated students and failed students have different values in the features, e.g. mother's job, father's job, dating or not.

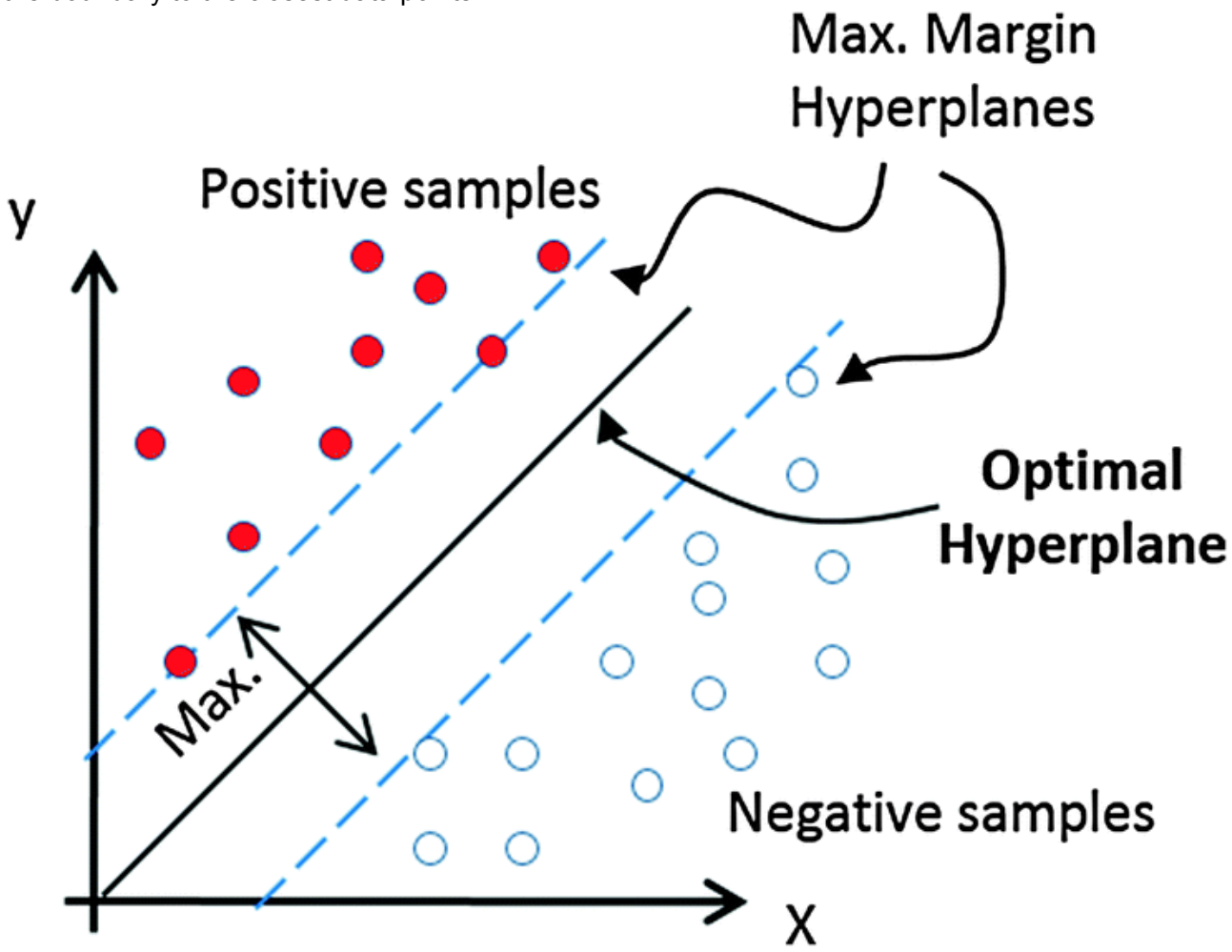
Then we use the model generated from the training set, which is basically two groups in our case, to predict whether some new students will graduate or not. We already know the new students' features, e.g. father's job, surfacing internet or not, etc. The features of the new students should be the same as the features in our model. Then according to the values in features of the new students, we are able to know which group it should below in our model. Then, the label of that group will be the label we will predict for the new student (will graduate or not)

One more thing I want to say, this is a general process for classification problems. Without considering the specific algorithms, all the classifiers should all try to solve the problem in the same way as described above. And from our testing (The section above), we choose SVC method.

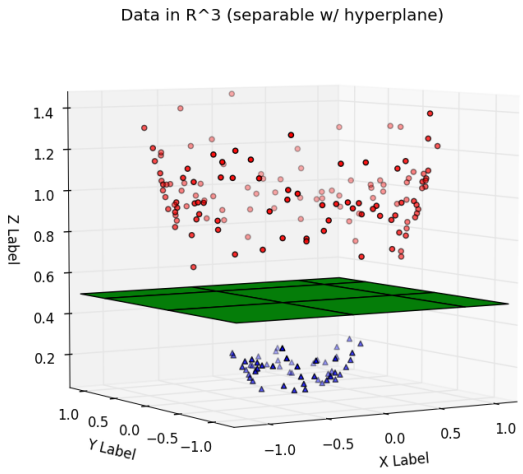
Supplementary Answer

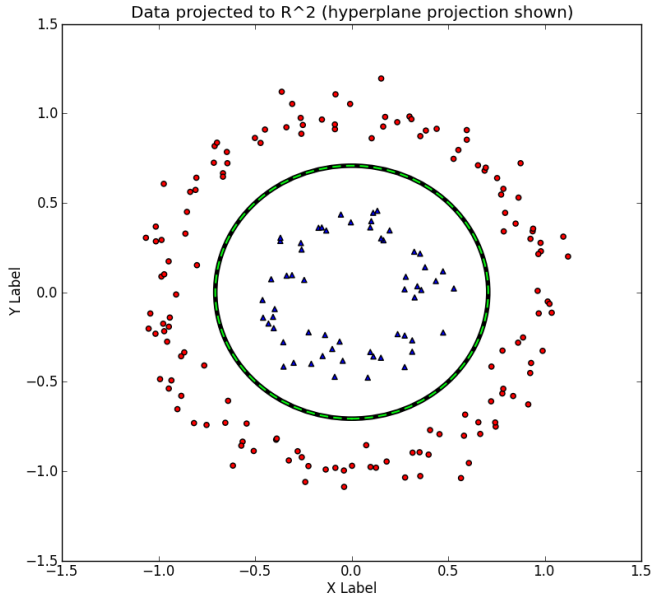
Reviewer asked me to give me a little bit more description about SVM method, in term of how it differs from other methods. Here is my explanation:

1. SVM builds a model using the dataset we have, by drawing a seperation boundary between different classes (in our case, students who graduated vs students who did not). SVM has the advantage to put the boundary at an optimal location, in order to maximize the margin, which is the distance from the boundary to the closest data points.



2. Sometimes it is difficult to draw the boundary in low dimentions to seperate different classes. We could use some tricks to design kernals to help drawing the boundary. The Example below shows that it is difficult to draw a straight line to seperate the labels in 2D domain, just using the X and Y coordinate values. However, by adding one more feature, $x^2 + y^2$, it is easy to draw a plane in the 3D dimention to seperate the labels.





3. Using the model (basically the location of the boundary) we build using SVM with the existing dataset, we can predict new data by putting them in the feature space and see which divided region it belongs to.

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/0.17/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 [31]:
```
# TODO: Import 'GridSearchCV' and 'make_scorer'
from sklearn.metrics import make_scorer,f1_score
from sklearn.grid_search import GridSearchCV

# classifier
from sklearn.svm import SVC

# TODO: Create the parameters list you wish to tune
parameters = [
  {'C': [0.1, 0.5, 1, 10], 'kernel': ['linear']},
  {'C': [0.1, 0.5, 1, 10],'degree':[2, 3], 'gamma': [0.5, 0.1, 0.01],
'kernel': ['poly']},
  {'C': [0.1, 0.5, 1, 10], 'gamma': [0.5, 0.1, 0.01], 'kernel': ['rbf']},
 ]
# TODO: Initialize the classifier
clf = SVC()

# 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 the scori
ng method
grid_obj = GridSearchCV(clf,parameters,scoring=f1_scorer)

# TODO: Fit the grid search object to the training data and find the optimal p
arameters
grid_obj = 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 tuning
print "Tuned model has a training F1 score of {:.4f}.".format(predict_labels(c
lf, X_train, y_train))
print "Tuned model has a testing F1 score of {:.4f}.".format(predict_labels(cl
f, X_test, y_test))
```

```
Made predictions in 0.0060 seconds.
Tuned model has a training F1 score of 0.9754.
Made predictions in 0.0020 seconds.
Tuned model has a testing F1 score of 0.8481.
```

In [30]:
```
print clf.get_params()
```

```
{'kernel': 'rbf', 'C': 1, 'verbose': False, 'probability': False, 'degree':
 3, 'shrinking': True, 'max_iter': -1, 'decision_function_shape': None, 'rand
om_state': None, 'tol': 0.001, 'cache_size': 200, 'coef0': 0.0, 'gamma': 0.1,
'class_weight': None}
```

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 best tuned model for SVC is

- 'rbf' kernel
- C=1
- gamma=0.1

The previous F1 score for default SVC mdoel is

> 0.8584 (train) 0.8462 (test)

The previous F1 score for default SVC mdoel is

> 0.9754 (train) 0.8481 (test)

While the training F1 score is improved significantly, the test F1 score is just slightly improved.

> **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.