student_intervention

April 18, 2016

1 Project 2: Supervised Learning

1.0.1 Building a Student Intervention System

1.1 Classification vs Regression

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

This is classification problem. We neede to classify students into two groups: needed or not needed early intervention (two-class classification), hence classification task.

1.2 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 [30]:
```
# Import libraries
import numpy as np
import pandas as pd
from sklearn.cross_validation import StratifiedShuffleSplit
from sklearn.cross_validation import KFold
from sklearn.utils import shuffle
import time
from sklearn.metrics import f1_score, make_scorer
from sklearn import tree
from sklearn import svm
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.cross_validation import train_test_split
from sklearn.grid_search import GridSearchCV
from sklearn.preprocessing import normalize
import random
```

In [9]:
```
#Read student data
student_data = pd.read_csv("student-data.csv")
print "Student data read successfully!"
# Note: The last column 'passed' is the target/label, all other are feature columns
```

```
Student data read successfully!
```

Now, can you find out the following facts about the dataset? - Total number of students - Number of students who passed - Number of students who failed - Graduation rate of the class (%) - Number of features

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

```
In [10]: # TODO: Compute desired values - replace each '?' with an appropriate expression/function call
         n_students = student_data.shape[0]
         n_features = student_data.shape[1] - 1
         n_passed = pd.value_counts(student_data.passed == "yes")[True]
         n_failed = pd.value_counts(student_data.passed == "yes")[False]
         grad_rate = float(n_passed)/(n_passed + n_failed)*100
         print "Total number of students: {}".format(n_students)
         print "Number of students who passed: {}".format(n_passed)
         print "Number of students who failed: {}".format(n_failed)
         print "Number of features: {}".format(n_features)
         print "Graduation rate of the class: {:.2f}%".format(grad_rate)
```

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

1.3 Preparing the Data

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

1.3.1 Identify feature and target columns

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

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

```
In [11]: # Extract feature (X) and target (y) columns
         feature_cols = list(student_data.columns[:-1])  # all columns but last are features
         target_col = student_data.columns[-1]  # last column is the target/label
         print "Feature column(s):-\n{}".format(feature_cols)
         print "Target column: {}".format(target_col)

         global X_all
         global y_all

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

```
Feature column(s):-
['school', 'sex', 'age', 'address', 'famsize', 'Pstatus', 'Medu', 'Fedu', 'Mjob', 'Fjob', 'reason', 'gu
Target column: passed

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

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

  absences
0        6
1        4
2       10
3        2
4        4

[5 rows x 30 columns]
```

1.3.2 Preprocess feature columns

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

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

These generated columns are sometimes called dummy variables, and we will use the `pandas.get_dummies()` function to perform this transformation.

```
In [12]: # Preprocess feature columns
         def preprocess_features(X):
             outX = pd.DataFrame(index=X.index)  # output dataframe, initially empty
             #print outX.columns[:]
             # Check each column
             for col, col_data in X.iteritems():
                 # If data type is non-numeric, try to replace all yes/no values with 1/0
                 if col_data.dtype == object:
                     col_data = col_data.replace(['yes', 'no'], [1, 0])
                 # Note: This should change the data type for yes/no columns to int

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

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

             return outX
         global X_all
         global y_all
         X_all = preprocess_features(X_all)
         y_all = preprocess_features(y_all)
         print "Processed feature columns ({}):-\n{}".format(len(X_all.columns), list(X_all.columns))
```

```
Processed feature columns (48):-
['school_GP', 'school_MS', 'sex_F', 'sex_M', 'age', 'address_R', 'address_U', 'famsize_GT3', 'famsize_LE3'
```

1.3.3 Split data into training and test sets

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

In [14]:
```
# First, decide how many training vs test samples you want
num_all = student_data.shape[0]  # same as len(student_data)
num_train = 300  # about 75% of the data
num_test = num_all - num_train

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


X_train, X_test, y_train, y_test = train_test_split(X_all, y_all,
                                                    test_size=num_test,
                                                    train_size=num_train,
                                                    stratify = y_all)
y_train = y_train.as_matrix().reshape((y_train.shape[0]))
y_test = y_test.as_matrix().reshape((y_test.shape[0]))

# X_train, y_train, X_test, y_test = next_batch(rs, train_size = num_train ,test_size = num_te
print "Training set: {} samples".format(X_train.shape[0])
print "Test set: {} samples".format(X_test.shape[0])

# Note: If you need a validation set, extract it from within training data
```

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

1.4 Training and Evaluating Models

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

- What is the theoretical O(n) time & space complexity in terms of input size?
- What are the general applications of this model? What are its strengths and weaknesses?
- Given what you know about the data so far, why did you choose this model to apply?
- Fit this model to the training data, try to predict labels (for both training and test sets), and measure the F1 score. Repeat this process with different training set sizes (100, 200, 300), keeping test set constant.

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

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

In [15]:
```
# Train a model

def train_classifier(clf, X_train, y_train):
    print "Training {}...".format(clf.__class__.__name__)
    start = time.time()
    clf.fit(X_train, y_train)
    end = time.time()
    print "Done!\nTraining time (secs): {:.3f}".format(end - start)
    return "{:.5f}".format(end - start)
```

```
# TODO: Choose a model, import it and instantiate an object

def create_classifier(type_of = "Tree", weights = None):
    if type_of == "Tree":
        return tree.DecisionTreeClassifier(class_weight=weights) #used parameter class_weight
    elif type_of == "SVM":
        return svm.SVC(class_weight=weights) #used parameter class_weight because dataset not
    elif type_of == "GrBoost":
        return GradientBoostingClassifier(n_estimators=100, learning_rate=.1, max_depth=3)
    else:
        raise ValueError("Classifier not found", type_of)


clf = create_classifier("Tree")
train_classifier(clf, X_train, y_train)
print clf  # you can inspect the learned model by printing it
#print
```

```
Training DecisionTreeClassifier...
Done!
Training time (secs): 0.005
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            presort=False, random_state=None, splitter='best')
```

In [16]:
```
# Predict on training set and compute F1 score
from sklearn.metrics import f1_score

def predict_labels(clf, features, target):
    print "Predicting labels using {}...".format(clf.__class__.__name__)
    start = time.time()
    y_pred = clf.predict(features)
    end = time.time()
    print "Done!\nPrediction time (secs): {:.5f}".format(end - start)
    return f1_score(target, y_pred), "{:.5f}".format(end - start)

train_f1_score, time_ = predict_labels(clf, X_train, y_train)
print "F1 score for training set: {:.3f} in {} sec".format(train_f1_score, time_)
```

```
Predicting labels using DecisionTreeClassifier...
Done!
Prediction time (secs): 0.00071
F1 score for training set: 1.000 in 0.00071 sec
```

In [17]:
```
# Predict on test data
print "F1 score for test set: {}".format(predict_labels(clf, X_test, y_test))
```

```
Predicting labels using DecisionTreeClassifier...
Done!
Prediction time (secs): 0.00059
F1 score for test set: (0.69421487603305776, '0.00059')
```

1.4.1 Decision Trees

Complexity

Decision Trees in general need:

$$O(n_{samples} \times n_{features} \times \log n_{samples}).$$

to construct balanced binary tree and query time:

$$O(\log n_{samples}).$$

For the project‘s problem complexity is:

```
In [18]: samples_ = 100
         print "Complexity for construction O({:d}) for {:d} samples".format(int(samples_*48*np.log2(sa
         print "Complexity for query O({:d}) for {:d} samples".format(int(np.log2(samples_)), samples_)
```

```
Complexity for construction O(31890) for 100 samples
Complexity for query O(6) for 100 samples
```

General applications

Decision-trees best suited to problems with following characteristics:

- Samples are represented by attribute-value pairs. Each feature takes on a small number of disjoint possible values (e.g., man, woman).
- Best for two possible output values.
- Problems with disjunctive descriptions.
- The training data may contain errors. Decision-tree learning methods are robust for errors.
- Training data may contain missing attribute data.

Strengths and weaknesses

Strengths:

- Simple to understand and to interpret, especially using graphic representation.
- White box - learned model will be explained using boolean logic.
- Data preparation not very hard. No needs data normalization, dummy and empty value filtering.
- Computation cost is relatively low - logarithmic for tree training and prediction.
- Possible validate model using statistical tests.

Weaknesses:

- Decision tree learners create biased trees if some classes dominate. Need balanced training data for every class or need equal number of samples for each class.
- Usually decision-tree learners generate overfitted models and for preventing that depth max and minimum number of samples at a leaf node are necessary.
- Small variations of data might result of completely different trees being generated.
- Learning optimal decision-tree is NP-problem. In practice heuristic methods usually used and that are not guarantee globally optimal tree (to avoid this multiple decision trees will be used with randomly sampled features and samples).

Dataset analysis

```
In [19]: def check_of_empty_values(data_set):
             #create empty Data frame for broken data (NaN or empty)
             return data_set.isnull().values.any()


         def get_labels_balance(data_set):
             #coutn labels in training or testing set and return label weights
             unique, counts = np.unique(data_set, return_counts=True)
             label_weights = {}
             for i in range(0, unique.shape[0]):
                 label_weights[unique[i]] = float(counts[i])/data_set.shape[0]
             return label_weights


         print "Is Student feature dataset contains NaN or empty values ?", check_of_empty_values(X_all)
         print "Is Student label dataset contains NaN or empty values ?", check_of_empty_values(y_all)
         print "Weights of data balance:"
         print "Training data:", get_labels_balance(y_train)
         print "Testing data:", get_labels_balance(y_test)
```

```
Is Student feature dataset contains NaN or empty values ? False
Is Student label dataset contains NaN or empty values ? False
Weights of data balance:
Training data: {0: 0.33, 1: 0.67}
Testing data: {0: 0.3263157894736842, 1: 0.6736842105263158}
```

Justification

Decision-Trees will be used for classification and regression problems with single and multi-variable output. Since DT is white box it is possible to explain prediction results and it may be useful for stuff.

Looking on the students data the DT also may be used for such problem because:

- most of attributes values are two-pared;
- predicting value belongs for the two classes;
- dataset specially was not prepared (excluding data type convertion to numbers and using dummies variables).

Although data set is not balanced this may mitigated by using label_weights parameter for classifier.

Training and prediction

```
In [24]: # Train and predict using different training set sizes
         def train_predict(clf, X_train, y_train, X_test, y_test):
             #print "------------------------------------------"
             #print "Training set size: {}".format(len(X_train))
             tr_weights = get_labels_balance(y_train)
             time_str = train_classifier(clf, X_train, y_train)
             tr_pred = predict_labels(clf, X_train, y_train)
             #print "F1 score for training set: {}".format(tr_pred)
             ts_pred = predict_labels(clf, X_test, y_test)
             #print "F1 score for test set: {}".format(ts_pred)
             return [{"Size":len(X_train), "Time":time_str, "F1_training":tr_pred, "F1_testing":ts_pred}]
```

```
#Create Pandas table
def append_value(frame, dict_values, i):
    if type(frame) is pd.DataFrame:
        new_frame = pd.DataFrame(dict_values, index=[i])
        return pd.concat([frame, new_frame])
    else:
        return pd.DataFrame(dict_values, index=[i])


# TODO: Run the helper function above for desired subsets of training data
# Note: Keep the test set constant
training_set_sizes = [100, 200, 300]
tree_table = None
i = 1
for set_size in training_set_sizes:
    #X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=num_test, tra
    #X_train, y_train, X_test, y_test = next_batch(rs, train_size=set_size, test_size=num_test

    # Create Decision-tree classifier and use label weights
    lab_weights = get_labels_balance(y_train[:set_size])
    clf = create_classifier("Tree", weights=lab_weights)
    train_time = train_classifier(clf, X_train[:set_size], y_train[:set_size])
    #print X_train[:set_size].shape
    f1_training, tr_time = predict_labels(clf, X_train[:set_size], y_train[:set_size])
    f1_testing, ts_time = predict_labels(clf, X_test, y_test)
    row = {"Size":set_size, "F1_training":f1_training, "F1_testing":f1_testing, "Time train":t
    tree_table = append_value(tree_table, row, i)
    i += 1

# Result of Decision Tree
print "\nDecision Tree training results\n", tree_table
```

```
Training DecisionTreeClassifier...
Done!
Training time (secs): 0.002
Predicting labels using DecisionTreeClassifier...
Done!
Prediction time (secs): 0.00028
Predicting labels using DecisionTreeClassifier...
Done!
Prediction time (secs): 0.00029
Training DecisionTreeClassifier...
Done!
Training time (secs): 0.002
Predicting labels using DecisionTreeClassifier...
Done!
Prediction time (secs): 0.00030
Predicting labels using DecisionTreeClassifier...
Done!
Prediction time (secs): 0.00026
Training DecisionTreeClassifier...
Done!
Training time (secs): 0.003
```

```
Predicting labels using DecisionTreeClassifier...
Done!
Prediction time (secs): 0.00035
Predicting labels using DecisionTreeClassifier...
Done!
Prediction time (secs): 0.00027

Decision Tree training results
   F1_testing  F1_training  Size Time test Time train
1    0.744526            1   100   0.00029    0.00163
2    0.781955            1   200   0.00026    0.00225
3    0.688000            1   300   0.00027    0.00348
```

Summary

Training time increase with approximately same rate as training size increase. Testing time fluctuating on the same level. Model for all training sets looks like overfitted.

1.4.2 Support Vector Machines

Complexity. The core of SVM is quadratic programming problem (QP), separating support vectors from the rest of the training data. Support Vector Machines for scipy implementation needs between

$$O(n_{features} \times n_{samples}^2)$$

and

$$O(n_{features} \times n_{samples}^3)$$

For the project's problem complexity is:

```
In [21]: samples_ = 100
         print "Complexity for construction O({:.3e}) for {:d} samples".format(int(48*samples_**2), samp
         print "Complexity for query O({:.3e}) for {:d} samples".format(int(48*samples_**3), samples_)
```

```
Complexity for construction O(4.800e+05) for 100 samples
Complexity for query O(4.800e+07) for 100 samples
```

General applications. Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection. Effective in high dimensional spaces. Still effective in cases where number of dimensions is greater than the number of samples. Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.

Strengths and weaknesses

Strengths:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

Weakness:

- If the number of features is much greater than the number of samples, the method is likely to give poor performances.
- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see Scores and probabilities, below).
- Support Vector Machine algorithms are not scale invariant.

Justification

- Number of features is relatively large compare to number of samples.
- SVM will be used for two-class problem.
- Prediction will use relatively small piece of memory and be quick- only support vectors are stored.
- SVM also will be used for classification problems.
- Kernel function can express domain knowledge.

Training and prediction

In [38]:
```
# SVM
training_set_sizes = [100, 200, 300]
svm_table = None
i = 1
for set_size in training_set_sizes:

    #X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=num_test, tra
    #X_train, y_train, X_test, y_test = next_batch(rs, train_size=set_size, test_size=num_test

    # Create Decision-tree classifier and use label weights
    lab_weights = get_labels_balance(y_train[:set_size])
    clf = create_classifier("SVM")


    train_time = train_classifier(clf, X_train[:set_size], y_train[:set_size])
    f1_training, tr_time = predict_labels(clf, X_train[:set_size], y_train[:set_size])
    f1_testing, ts_time = predict_labels(clf, X_test, y_test)
    row = {"Size":set_size, "F1_training":f1_training, "F1_testing":f1_testing, "Time train":t
    svm_table = append_value(svm_table, row, i)
    i += 1

# SVM training results
print "\nSVM training results\n", svm_table
```

```
Training SVC...
Done!
Training time (secs): 0.003
Predicting labels using SVC...
Done!
Prediction time (secs): 0.00182
Predicting labels using SVC...
Done!
Prediction time (secs): 0.00159
Training SVC...
Done!
Training time (secs): 0.007
Predicting labels using SVC...
Done!
```

```
Prediction time (secs): 0.00523
Predicting labels using SVC...
Done!
Prediction time (secs): 0.00260
Training SVC...
Done!
Training time (secs): 0.014
Predicting labels using SVC...
Done!
Prediction time (secs): 0.01070
Predicting labels using SVC...
Done!
Prediction time (secs): 0.00362

SVM training results
   F1_testing  F1_training  Size  Time test  Time train
1    0.800000     0.901961   100    0.00159     0.00259
2    0.812903     0.855305   200    0.00260     0.00721
3    0.818182     0.845666   300    0.00362     0.01406
```

Summary. The training time increase very rapidly with increasing training set. See calc. below:

```
In [34]: print "Times {:.3f}".format(float(svm_table["Time train"].iloc[2])/float(svm_table["Time train
```

```
Times 6.963
```

Testing time increase but with less rate as training time.

1.4.3 Gradient Boosting

Complexity. The algorithm for Boosting Trees evolved from the application of boosting methods to regression trees. The general idea is to compute a sequence of (very) simple trees, where each successive tree is built for the prediction residuals of the preceding tree. The complexity of Gradient Boosting depends on number of decision trees and their depth and features number.

General applications. Gradient Boosted Regression Trees (GBRT) is a generalization of boosting to arbitrary differentiable loss functions. GBRT is an accurate and effective off-the-shelf procedure that can be used for both regression and classification problems.

Strengths and weakness

Strengths:

- Natural handling of data of mixed type (= heterogeneous features)
- Predictive power
- Robustness to outliers in output space (via robust loss functions)
- Robustness to data scaling

Weakness:

- Scalability, due to the sequential nature of boosting it can hardly be parallelized.

Justifications

- GradientBoostingClassifier supports both binary and multi-label classification
- Prediction time relatively low
- Supports mixed type of features and not needed data normalization

Training and prediction

In [28]:
```
training_set_sizes = [100, 200, 300]
boost_table = None
i = 1
for set_size in training_set_sizes:
    #X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=num_test, tra
    #X_train, y_train, X_test, y_test = next_batch(rs, train_size=set_size, test_size=num_test

    clf = create_classifier("GrBoost")


    train_time = train_classifier(clf, X_train[:set_size], y_train[:set_size])
    f1_training, tr_time = predict_labels(clf, X_train[:set_size], y_train[:set_size])
    f1_testing, ts_time = predict_labels(clf, X_test, y_test)
    row = {"Size":set_size, "F1_training":f1_training, "F1_testing":f1_testing, "Time train":t
    boost_table = append_value(boost_table, row, i)
    i += 1

# Gradient Boosting training results
print "\nGradient Boosting training results\n", boost_table
```

```
Training GradientBoostingClassifier...
Done!
Training time (secs): 0.107
Predicting labels using GradientBoostingClassifier...
Done!
Prediction time (secs): 0.00099
Predicting labels using GradientBoostingClassifier...
Done!
Prediction time (secs): 0.00083
Training GradientBoostingClassifier...
Done!
Training time (secs): 0.146
Predicting labels using GradientBoostingClassifier...
Done!
Prediction time (secs): 0.00143
Predicting labels using GradientBoostingClassifier...
Done!
Prediction time (secs): 0.00080
Training GradientBoostingClassifier...
Done!
Training time (secs): 0.186
Predicting labels using GradientBoostingClassifier...
Done!
Prediction time (secs): 0.00174
Predicting labels using GradientBoostingClassifier...
Done!
Prediction time (secs): 0.00079
```

```
Gradient Boosting training results
   F1_testing  F1_training  Size Time test Time train
1    0.794118     1.000000   100   0.00083    0.10686
2    0.757576     0.996255   200   0.00080    0.14623
3    0.733813     0.970874   300   0.00079    0.18613
```

Summary Training time slowly increase as increasing training set size. Prediction time fluctuates on same level.

1.4.4 Summary of algorithms

In [29]:
```
print "Decision Tree"
print tree_table

print "\nSVM"
print svm_table

print "\nGradient boosting"
print boost_table
```

```
Decision Tree
   F1_testing  F1_training  Size Time test Time train
1    0.744526            1   100   0.00029    0.00163
2    0.781955            1   200   0.00026    0.00225
3    0.688000            1   300   0.00027    0.00348

SVM
   F1_testing  F1_training  Size Time test Time train
1    0.800000     0.901961   100   0.00198    0.00327
2    0.812903     0.855305   200   0.00265    0.00738
3    0.818182     0.845666   300   0.00362    0.01435

Gradient boosting
   F1_testing  F1_training  Size Time test Time train
1    0.794118     1.000000   100   0.00083    0.10686
2    0.757576     0.996255   200   0.00080    0.14623
3    0.733813     0.970874   300   0.00079    0.18613
```

1.5 Choosing the Best Model

- Based on the experiments you performed earlier, in 1-2 paragraphs explain to the board of supervisors what single model you chose as the best model. Which model is generally the most appropriate based on the available data, limited resources, cost, and performance?
- In 1-2 paragraphs explain to the board of supervisors in layman's terms how the final model chosen is supposed to work (for example if you chose a Decision Tree or Support Vector Machine, how does it make a prediction).
- Fine-tune the model. Use Gridsearch with at least one important parameter tuned and with at least 3 settings. Use the entire training set for this.
- What is the model's final F1 score?

1.5.1 Most appropriate model

The SVM was choosed as most appropriate model for the particular problem. > DecisionTrees model looks like overfitted on the given data and size of it. Although training and testing time was lowest comparing to other models. DS model was rejected - on given data it overfitting and can't give general model.

> Gradient boosting for the given problem may be used, but there are no more data to construct general model. For example, of the size 300 model became overfitted. Although Gradient Boosting is white box model and this can be used for staff. Gradient boosting was taken for training greater time as other models, although testing time is lowest. Gradient boosting was rejected - more data will needed and F1 accuracy score less than for SVM.
>
> Compare to the available data and resorces the SVM is more appropriate model and may construct general model for the particular problem.

1.5.2 SVM prediction in Layman's terms

SVM is good choice for the Student intervention problem, where hight dimensionality of the data (large number of descriptive fields exists). In future this dimensionality probably will increase (especially with using e-learning platform and taking into account students activities) and SVM has strong side of classifying such data.

Main idea in a SVM is to separate data with boundaries which best separates data on different classes. For example in our problem we need to classify students into two classes: passed and not passed exam. **In the learning stage** SVM try to find best way how to separate given data for the two classes and builds mathematical model for furfer predictions.

The best way to understand leaning of SVM is graphical representation:

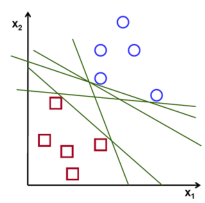

Figure 1: "SVM training"

Where x1 and x2 are description information about students. For example, x1 is failures (ranged) and x2 - family status (also ranged). Circles represents students who passed exam and squares - who are not. SVN trying to find (in this example) linear separator (line) which best separates our classes of students.

What means best separator? Here answer on the next figure:

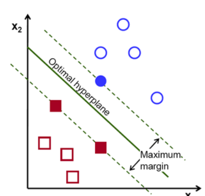

Figure 2: "Bondaries of two classes"

Best separator is two lines (boundaries) on circles and squares (in linear case) of the two classes which are given maximal distance between lines. Circles and squares on boundaries are closes to the middle line between boundaries and are named **support vectors**.

After SVM find maximal boundaries then learning process finish and SVM is ready for using.

For some problems not possible to find best separator (called **kernel function**) using linear function. In such situations will be used non-linear separator. Example you can see in Figure 3.

After a new student come to learning we have a lot of data about him/her for prediction, some data not available and we leave it to default preferred values. Providing such prepared data to SVM model it applying

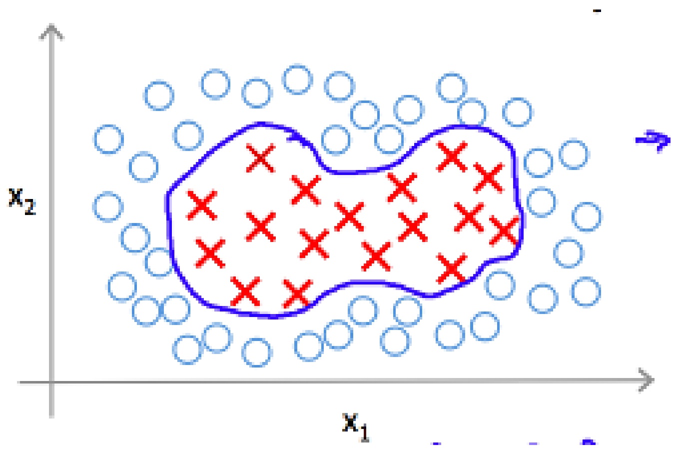

Figure 3: “Non-linear kernel”

kernel function (linear or nonlinear) using learned parameters (weights and biases) and student data vector (prepared student data fields values). As a result kernel return value -1 or +1. For example, +1 specifying class of students who are passed exam and -1 accordingly who are not. Such sequence not strict.

As mentioned before, in the learning stage classes divided by decision boundary (also named hyperplane) where each side points labeled differently to other side with value +1 or -1. Such values for classes are not choised arbitrary, but for mathematically divide data into two classes. For example, circles in Figure 1 in SVM model has label -1 and squares +1.

1.5.3 Tuning SVM

As main three criteria were used for tuning SVM (SVC function):

- C - penalty parameter. It corresponds to regularize more the estimation if dataset is noisy (C value will be increased). Default=1.0.
- Kernel function: rbf (default), linear, poly, sigmoid and custom.
- Tolerance - stopping criterion tolerance (default=1e-3).

```
In [54]: c_range = np.arange(0.8, 1.9,.05)
         tol_range = np.arange(0.0001, 0.01, 0.0005)
         gamma =

         # parameters used with GridSearch for SVM
         params_dic = {"C":c_range, "kernel": ['rbf', 'linear', 'poly', 'sigmoid'], "tol" : tol_range}

         # prepare CV
         cv = StratifiedShuffleSplit(y_train, n_iter = 10, random_state = 42)

         # create SVM classifier and grid_search, define scorer
         scorer = make_scorer(f1_score)
         clf = svm.SVC()
         grid_search_obj = GridSearchCV(estimator=clf, param_grid=params_dic, scoring=scorer, iid=True,
                                        cv=cv, verbose=True, n_jobs=16)
         # Fit data with GridSearch
         grid_search_obj.fit(X_train, y_train)

         #get best estimator
         best_estimator = grid_search_obj.best_estimator_
```

```
print "Best parameters:", best_estimator
```

```
Fitting 10 folds for each of 1760 candidates, totalling 17600 fits

[Parallel(n_jobs=16)]: Done  80 tasks      | elapsed:    1.4s
[Parallel(n_jobs=16)]: Done 559 tasks      | elapsed:   10.2s
[Parallel(n_jobs=16)]: Done 978 tasks      | elapsed:   13.9s
[Parallel(n_jobs=16)]: Done 1678 tasks      | elapsed:   26.7s
[Parallel(n_jobs=16)]: Done 2390 tasks      | elapsed:   44.2s
[Parallel(n_jobs=16)]: Done 3490 tasks      | elapsed:  1.0min
[Parallel(n_jobs=16)]: Done 4990 tasks      | elapsed:  1.5min
[Parallel(n_jobs=16)]: Done 6115 tasks      | elapsed:  1.9min
[Parallel(n_jobs=16)]: Done 7884 tasks      | elapsed:  2.5min
[Parallel(n_jobs=16)]: Done 9608 tasks      | elapsed:  3.0min
[Parallel(n_jobs=16)]: Done 11350 tasks      | elapsed:  3.6min
[Parallel(n_jobs=16)]: Done 12907 tasks      | elapsed:  4.1min
[Parallel(n_jobs=16)]: Done 14844 tasks      | elapsed:  4.9min
[Parallel(n_jobs=16)]: Done 17559 tasks      | elapsed:  6.0min
[Parallel(n_jobs=16)]: Done 17600 out of 17600 | elapsed:  6.0min finished

Best parameters: SVC(C=0.8000000000000004, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.0001, verbose=False)
```

1.5.4 F1 score for tuned SVM

In [55]:
```
# Predict data
y_pred = best_estimator.predict(X_test)
result = f1_score(y_test, y_pred)
print "F1 score: {:.3f}".format(result)
```

```
F1 score: 0.808
```