Assignment 3 - Part 2: Perceptrons, SVMs, Multiclass Classification, Decision Boundaries (60 points)

In this part of the assignment, you'll work with two other classification algorithms - Perceptrons and Support Vector Machines. You'll also have a chance to understand the mechanics behind the multiclass classification approach used when there are more than one class in scikit-learn. Finally, you'll use some example code in scikit-learn to understand the similarities and differences between different classification techniques.

In [16]:
```
## Preliminaries

#Show plots in the notebook
%matplotlib inline

from sklearn import datasets, preprocessing, cross_validation, feature_ex
from sklearn import linear_model, svm, metrics, ensemble
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import urllib2

# Helper functions
def folds_to_split(data,targets,train,test):
    data_tr = pd.DataFrame(data).iloc[train]
    data_te = pd.DataFrame(data).iloc[test]
    labels_tr = pd.DataFrame(targets).iloc[train]
    labels_te = pd.DataFrame(targets).iloc[test]
    return [data_tr, data_te, labels_tr, labels_te]
```

The (Kernel) Perceptron

The perceptron algorithm is one of the early, classic machine learning algorithms. It has a number
advantages: it's fast since the learning updates don't require any costly matrix inversions or linear
solvers, the parameters can be updated online, and if the target concept can be expressed as a hy
that cleanly separates the classes, the perceptron algorithm will provably find the concept. Of cour
there isn't a separating hyperplane, perceptron will never converge.

Perceptron learning (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html#sklearn.linear_model.Pe
is also found in the scikit-learn linear_model module. The parameters are similar to logistic regress

can specify regularization methods and weights. In addition, there's a `n_iter` parameter that cont
many times the perceptron algorithm will run through the data. This is important since without setti
limit, the perceptron algorithm runs until convergence, which might be never!

Question 1: Perceptron learning (15 points)

1. Load the heart dataset from Assignment 1, Part 3. Perform the steps necessary to get the dataset ready for classification. You should convert the label field (HeartDisease) to binary, with values greater than 0 mapped to 1 (see Q3.4). As a guide, you might refresh your memory of the steps we used in Assignment 1 and look at the preparatory steps for the census dataset for Question 1.2. List the data processing steps you completed.
2. Create a 10-fold cross-validation experiment using `random_state=20160202`. Use the Perceptron to classify the heart data, using the values 1, 3, 5, 10, 20, and 100 for `n_iter`. Report the mean accuracy for each of the settings.
3. Transform the attributes using the `PolynomialFeatures` function from the `preprocessing` module. Repeat the 10-fold experiment, this time using no regularization, L1 regularization, and L2 regularization and varying the `alpha` parameter from 0.0001, 0.001, 0.01, and 0.1. Report the mean accuracy for each experiment (total of 12 numbers). What trends do you observe?

Answers 1:

1.

Preprocessing steps:
1) Replace values for HeartDisease > 0 to 1
2) Drop missing values
3) Standardize the data using the StandardScaler

2.

Perceptron Iterations: 1
Accuracy 0.784138

Perceptron Iterations: 3
Accuracy 0.789483

Perceptron Iterations: 5
Accuracy 0.792414

Perceptron Iterations: 10
Accuracy 0.789684

```
Perceptron Iterations: 20
Accuracy 0.780598

Perceptron Iterations: 100
Accuracy 0.785153
```

3.

When we vary the alpha parameter, we essentially are seeing the effects of the bias-variance tradeoff.
High alpha fixes high variance, thus preventing overfitting.
Low alpha fixes high bias, thus preventing underfitting.

When no regularization is applied, we don't see any patterns in mean accuracy changes. This is understandable because alpha only applies when regularization is used.

When we use L1 regularization, as alpha increases, the accuracy drops much more significantly than when we used L2 normalization. With L1 there was a drop of ~4% in accuracy whereas with L2 there was a drop of ~2% accuracy.

```
Penalty: l1 and Alpha: 0.0001
Accuracy 0.724138


Penalty: l1 and Alpha: 0.001
Accuracy 0.73931


Penalty: l1 and Alpha: 0.01
Accuracy 0.750115


Penalty: l1 and Alpha: 0.1
Accuracy 0.719569


Penalty: l2 and Alpha: 0.0001
Accuracy 0.725885


Penalty: l2 and Alpha: 0.001
Accuracy 0.728391
```

```
Penalty: l2 and Alpha: 0.01
Accuracy 0.727767


Penalty: l2 and Alpha: 0.1
Accuracy 0.713822


Penalty: None and Alpha: 0.0001
Accuracy 0.715709


Penalty: None and Alpha: 0.001
Accuracy 0.717218


Penalty: None and Alpha: 0.01
Accuracy 0.718454


Penalty: None and Alpha: 0.1
Accuracy 0.719483
```

In [17]:
```
# Part 1 Preprocessing

#let's load the data
heart_data = urllib2.urlopen("http://archive.ics.uci.edu/ml/machine-learn
heart = pd.read_csv(heart_data,
                    quotechar='"',
                    skipinitialspace=True,
                    names=['Age',
                           'Sex',
                           'ChestPainType',
                           'RestingBP',
                           'Cholesterol',
                           'FastingBloodSugar',
                           'RestingECG',
                           'MaxHeartRate',
                           'ExerciseInducedAngina',
                           'STExerciseDepression',
                           'STExercisePeakSlope',
                           'FlouroscopyVessels',
                           'Thalassemia',
                           'HeartDisease'], na_values="?")

# Replace values for HeartDisease > 0 to 1
heart.loc[heart['HeartDisease'] > 0, 'HeartDisease'] = 1

# Drop missing values
heart = heart.dropna()

# Extract labels and data values
heart_data_not_normalized = heart[['Age', 'Sex', 'ChestPainType', 'Restin
heart_target = heart['HeartDisease']

# standardize the data using Standard Scaler
std_scaler = preprocessing.StandardScaler()
heart_data = pd.DataFrame(std_scaler.fit_transform(heart_data_not_normali
```

In [18]:
```
# Part 2 Perceptron iterations varying
foldnum = 0
fold_results = pd.DataFrame()

iterations = [1, 3, 5, 10, 20, 100]

for iteration in range(len(iterations)):

    print "Perceptron Iterations: ", iterations[iteration]

    for train, test in cross_validation.KFold(len(heart_data), n_folds=10
        foldnum+=1
        [heart_tr_data, heart_te_data,
         heart_tr_target, heart_te_target] = folds_to_split(heart_data, h

        perceptron = linear_model.Perceptron(n_iter=iterations[iteration]
        perceptron.fit(heart_tr_data, np.reshape(heart_tr_target.values,[
#         lgr.fit(adult_tr_data, np.reshape(adult_tr_target.values,[len(a


        fold_results.loc[foldnum, 'Accuracy'] = perceptron.score(heart_te

    #And compute the mean error across folds:
    print fold_results.mean()
```

```
Perceptron Iterations:  1
Accuracy    0.784138
dtype: float64
Perceptron Iterations:  3
Accuracy    0.789483
dtype: float64
Perceptron Iterations:  5
Accuracy    0.792414
dtype: float64
Perceptron Iterations:  10
Accuracy    0.789684
dtype: float64
Perceptron Iterations:  20
Accuracy    0.780598
dtype: float64
Perceptron Iterations:  100
Accuracy    0.785153
dtype: float64
```

In [19]:
```
# Part 3 Polynomial Features and Perceptron variations
foldnum = 0
fold_results = pd.DataFrame()

# Get the Polynomial Features of the heart data
heart_polynomial = pd.DataFrame(preprocessing.PolynomialFeatures().fit_tr

alphas = [0.0001, 0.001, 0.01, 0.1]
regularization = ['l1', 'l2', 'None']

for penalty in range(len(regularization)):
    for alpha in range(len(alphas)):
        print "Penalty: " + str(regularization[penalty]) + " and Alpha: "

        for train, test in cross_validation.KFold(len(heart_polynomial),
            foldnum+=1
            [heart_tr_data, heart_te_data,
             heart_tr_target, heart_te_target] = folds_to_split(heart_pol

            perceptron = linear_model.Perceptron(alpha=alphas[alpha], pen
            perceptron.fit(heart_tr_data, np.reshape(heart_tr_target.valu

            fold_results.loc[foldnum, 'Accuracy'] = perceptron.score(hear

        print fold_results.mean()
        print "------\n"
```

```
Penalty: l1 and Alpha: 0.0001
Accuracy    0.724138
dtype: float64
------

Penalty: l1 and Alpha: 0.001
Accuracy    0.73931
dtype: float64
------

Penalty: l1 and Alpha: 0.01
Accuracy    0.750115
dtype: float64
------

Penalty: l1 and Alpha: 0.1
Accuracy    0.719569
dtype: float64
------

Penalty: l2 and Alpha: 0.0001
Accuracy    0.725885
dtype: float64
------

Penalty: l2 and Alpha: 0.001
Accuracy    0.728391
dtype: float64
------

Penalty: l2 and Alpha: 0.01
Accuracy    0.727767
dtype: float64
------

Penalty: l2 and Alpha: 0.1
Accuracy    0.713822
dtype: float64
------

Penalty: None and Alpha: 0.0001
Accuracy    0.715709
dtype: float64
------

Penalty: None and Alpha: 0.001
Accuracy    0.717218
dtype: float64
------

Penalty: None and Alpha: 0.01
Accuracy    0.718454
dtype: float64
------
```

```
Penalty: None and Alpha: 0.1
Accuracy    0.719483
dtype: float64
------
```

Support Vector Machines

While the Perceptron finds *a* separating hyperplane, support vector machines try to find a *good* separating hyperplane. They do this by trying to choose the hyperplane to maximize the distance to the nearest training instances for each class. The consequence of this algorithmic feature is that SVMs are often more generalizable.

Another attractive feature of SVMs is that they can operate in an expanded feature space without incurring the computational overheads required in other algorithms such as the Perceptron and logistic regression. The function used to expand the input features to the larger feature space is called a *kernel* and in the question below you will experiment with different kernels.

For now, let's look at some of the features of the SVM implementation in scikit-learn.

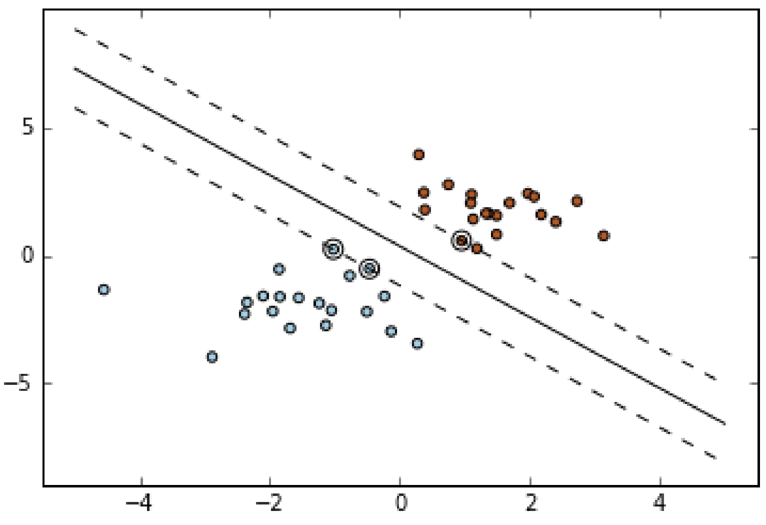
In [20]:
```
#load data
iris = datasets.load_iris()
#make a train-test split
[iris_tr_data, iris_te_data,
 iris_tr_labels, iris_te_labels] = cross_validation.train_test_split(iris
#create the SVM with a simple, linear kernel
iris_svm = svm.SVC(kernel='linear', random_state=20160202)
#learn the SVM
iris_svm.fit(iris_tr_data, iris_tr_labels)
#look at the support vectors
print "Number of support vectors for each class", iris_svm.n_support_
print iris_svm.support_vectors_
```

```
Number of support vectors for each class [ 3 11 11]
[[ 4.8  3.4  1.9  0.2]
 [ 5.1  3.3  1.7  0.5]
 [ 4.5  2.3  1.3  0.3]
 [ 6.3  3.3  4.7  1.6]
 [ 6.5  2.8  4.6  1.5]
 [ 5.1  2.5  3.   1.1]
 [ 6.7  3.   5.   1.7]
 [ 6.   2.7  5.1  1.6]
 [ 6.1  2.9  4.7  1.4]
 [ 6.2  2.2  4.5  1.5]
 [ 6.3  2.5  4.9  1.5]
 [ 6.   2.9  4.5  1.5]
 [ 5.6  3.   4.5  1.5]
 [ 6.1  3.   4.6  1.4]
 [ 7.2  3.   5.8  1.6]
 [ 6.   2.2  5.   1.5]
 [ 6.3  2.7  4.9  1.8]
 [ 6.3  2.5  5.   1.9]
 [ 6.1  3.   4.9  1.8]
 [ 5.9  3.   5.1  1.8]
 [ 4.9  2.5  4.5  1.7]
 [ 6.2  2.8  4.8  1.8]
 [ 6.   3.   4.8  1.8]
 [ 6.3  2.8  5.1  1.5]
 [ 6.5  3.2  5.1  2. ]]
```

To get a more visual representation of what's going on, I've adapted some code from the scikit-learn documentation (http://scikit-learn.org/stable/auto_examples/svm/plot_separating_hyperplane.html) showing the hyperplane SVM learns. The code below generates some random data, learns an SVM, plots the separating hyperplane, identifies the support vectors with a circle and shows the "margin" - the buffer around the hyperplane, which is what the SVM maximizes.

In [21]:
```
# we create 40 separable points
np.random.seed(0)
X = np.r_[np.random.randn(20, 2) - [2, 2], np.random.randn(20, 2) + [2, 2
Y = [0] * 20 + [1] * 20

# fit the model
clf = svm.SVC(kernel='linear')
clf.fit(X, Y)
print "Number of support vectors for each class:", clf.n_support_
print "Support vectors:", clf.support_vectors_

# get the separating hyperplane
w = clf.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(-5, 5)
yy = a * xx - (clf.intercept_[0]) / w[1]

# plot the parallels to the separating hyperplane that pass through the
# support vectors
b = clf.support_vectors_[0]
yy_down = a * xx + (b[1] - a * b[0])
b = clf.support_vectors_[-1]
yy_up = a * xx + (b[1] - a * b[0])

# plot the line, the points, and the nearest vectors to the plane
plt.plot(xx, yy, 'k-')
plt.plot(xx, yy_down, 'k--')
plt.plot(xx, yy_up, 'k--')

plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
            s=80, facecolors='none')
plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.Paired)

plt.axis('tight')
plt.show()
```

```
Number of support vectors for each class: [2 1]
Support vectors: [[-1.02126202  0.2408932 ]
 [-0.46722079 -0.53064123]
 [ 0.95144703  0.57998206]]
```



Question 2: Support Vector Machines (15 points)

1. Use SVMs to classify the heart data, using the same setup as Question 1 (including 10-fold cross-validation. Switch between different kernel functions: 'linear', 'poly', 'rbf', and 'sigmoid'.
    - Report the mean accuracy of each. Which performs best? How does this compare to the Perceptron's performance?
    - How many support vectors does each kernel use?
2. Use SVMs for regression on the diabetes dataset, again using the same cross-validation setup as demonstrated in Part 1. Again, try each of the four different kernels and compare their performance. Report the mean accuracy for each and compare it to linear regression

Answers 2:

1.

Really there's not one best performing kernel function. All, linear, polynomial and rbg kernels give us a very accurate data with accuracy of about 82.5%. The only kernel function that did not perform as well was the sigmoid. Furhtermore, all three really good kernels outperformed the best Perceptron's performance of close to 79%.
a) Mean accuracies:
Kernel Function: linear
Accuracy 0.824713

Kernel Function: poly
Accuracy 0.826494

Kernel Function: rbf

Accuracy 0.8259

Kernel Function: sigmoid
Accuracy 0.754138

b.Number of support vectors for each kernel:
Linear [51 50]
Poly [68 64]
RBF [71 68]
Sigmoid [84 82]

2.

As expected, the Linear Kernel function performs the best between the different support vector kernels. But the accuracy of the linear kernel svr is still lower than the normal regression function we used in part 1 by 6%

Linear Regression: R^2: 0.533716155537

Kernel Function: linear
Accuracy 0.477077

Kernel Function: poly
Accuracy 0.217649

Kernel Function: rbf
Accuracy 0.304249

Kernel Function: sigmoid
Accuracy 0.21774

In [22]:
```
# Part 1 SVM for heart data
foldnum = 0
fold_results = pd.DataFrame()

kernels = ['linear', 'poly', 'rbf', 'sigmoid']

for kernel in range(len(kernels)):

    print "Kernel Function: ", kernels[kernel]

    for train, test in cross_validation.KFold(len(heart_data), n_folds=10
        foldnum+=1
        [heart_tr_data, heart_te_data,
         heart_tr_target, heart_te_target] = folds_to_split(heart_data, h

        #create the SVM with a simple, linear kernel
        heart_svm = svm.SVC(kernel=kernels[kernel], random_state=20160202

        #learn the SVM
        heart_svm.fit(heart_tr_data, np.reshape(heart_tr_target.values,[l


        fold_results.loc[foldnum, 'SVMs1'] = heart_svm.n_support_[0]
        fold_results.loc[foldnum, 'SVMs2'] = heart_svm.n_support_[1]
        fold_results.loc[foldnum, 'Accuracy'] = heart_svm.score(heart_te_

        #look at the support vectors
        print "Number of support vectors for each class", heart_svm.n_sup
    print fold_results.mean()
```

```
Kernel Function:  linear
Number of support vectors for each class [49 51]
Number of support vectors for each class [47 48]
Number of support vectors for each class [52 51]
Number of support vectors for each class [53 51]
Number of support vectors for each class [51 51]
Number of support vectors for each class [54 56]
Number of support vectors for each class [49 46]
Number of support vectors for each class [50 45]
Number of support vectors for each class [53 50]
Number of support vectors for each class [53 51]
SVMs1       51.100000
SVMs2       50.000000
Accuracy     0.824713
dtype: float64
Kernel Function:  poly
Number of support vectors for each class [82 76]
Number of support vectors for each class [84 82]
Number of support vectors for each class [84 78]
Number of support vectors for each class [87 79]
Number of support vectors for each class [83 77]
Number of support vectors for each class [90 82]
Number of support vectors for each class [85 76]
Number of support vectors for each class [82 76]
Number of support vectors for each class [86 80]
Number of support vectors for each class [84 81]
SVMs1       67.900000
SVMs2       64.350000
Accuracy     0.826494
dtype: float64
Kernel Function:  rbf
Number of support vectors for each class [77 76]
Number of support vectors for each class [76 73]
Number of support vectors for each class [78 78]
Number of support vectors for each class [77 80]
Number of support vectors for each class [75 74]
Number of support vectors for each class [77 80]
Number of support vectors for each class [72 73]
Number of support vectors for each class [75 73]
Number of support vectors for each class [78 74]
Number of support vectors for each class [76 76]
SVMs1       70.633333
SVMs2       68.133333
Accuracy     0.825900
dtype: float64
Kernel Function:  sigmoid
Number of support vectors for each class [120 120]
Number of support vectors for each class [124 124]
Number of support vectors for each class [118 118]
Number of support vectors for each class [126 126]
Number of support vectors for each class [126 126]
Number of support vectors for each class [126 126]
Number of support vectors for each class [121 121]
Number of support vectors for each class [122 122]
Number of support vectors for each class [121 121]
Number of support vectors for each class [129 129]
SVMs1       83.800000
SVMs2       81.925000
Accuracy     0.754138
dtype: float64
```

In [23]:

```
# SVMs for diabetes data
diabetes = datasets.load_diabetes();
# Put it into pandas DataFrames
diabetes_data_df = pd.DataFrame(diabetes.data);
diabetes_target_df = pd.DataFrame(diabetes.target)

foldnum = 0
fold_results = pd.DataFrame()

kernels = ['linear', 'poly', 'rbf', 'sigmoid']

for kernel in range(len(kernels)):

    print "Kernel Function: ", kernels[kernel]

    for train, test in cross_validation.KFold(len(diabetes_data_df), n_fo
        foldnum+=1
        [diabetis_tr_data, diabetis_te_data,
         diabetis_tr_target, diabetis_te_target] = folds_to_split(diabete

        #create the SVM with a simple, linear kernel
        diabetis_svm = svm.SVR(kernel=kernels[kernel], C=1e3)

        #learn the SVM
        diabetis_svm.fit(diabetis_tr_data, np.reshape(diabetis_tr_target.

        fold_results.loc[foldnum, 'Accuracy'] = diabetis_svm.score(diabet
    print fold_results.mean()
```

```
Kernel Function:  linear
Accuracy    0.477077
dtype: float64
Kernel Function:  poly
Accuracy    0.217649
dtype: float64
Kernel Function:  rbf
Accuracy    0.304249
dtype: float64
Kernel Function:  sigmoid
Accuracy    0.21774
dtype: float64
```

Multiclass Classification

Using regression or SVM for classification is fairly straightforward when you have a binary split: you learn the line that separates the two classes and then round the response value. However, if there are multiple classes the process is less straightforward. What should you do?

There are two main strategies for multiclass classification with binary classifiers: one-versus-all classification and pairwise classification. In one-versus-all (aka one-versus-rest), you train one classifier for each class. The positive instances (1s) are those which have the label, while the negative labels (0s) are all of the other instances, regardless of which class they belong to. To make predictions, you run all C classifiers (assuming C different classes) and output the label that corresponds to the highest-valued classifier.

In pairwise classification (aka one-vs-one) you learn a classifier for each pair of classes (C-choose-2) classes. Each classifier is trained only on the data from the corresponding classes, and the data from other classes is excluded. When it's time to make a prediction, you feed the test instance to all of the different classifiers (C-choose-2 of them!). Each time a particular class label is output, that label gets a point. At the end, the label with the most points is the one output by the predictor.

These two primary multiclass approaches are in the multiclass module (http://scikit-learn.org/stable/modules/multiclass.html) of scikit-learn. You should read the documentation to understand how they work. There are other, more esoteric approaches to multiclass classification (like making a decision tree with each decision node filtering out half the classes), but we won't discuss them here.

In [24]:
```
#Generate some data for this question
gen_data, gen_labels = datasets.make_classification(n_classes=4, n_sample
                                                    n_clusters_per_class = 3, n_inform
```

Question 3: Multiclass classification (15 points)

1. Learn a `OneVsRestClassifier` for the generated dataset above with 10-fold cross-validation, using Logistic Regression as the classifier. Print out the list of classifiers (estimators) and apply the first estimator to the data. Report the mean accuracy across folds using the one-versus-all method of multiclass classification
2. Learn a `OneVsOneClassifier` for the generated dataset above with 10-fold cross-validation, using Logistic Regression as the classifier. Print out the list of classifiers (estimators) and apply the first estimator to the data. Report the mean accuracy across folds using pairwise multiclass classification.

Answers 3:

1.

OneVsRest Classifier:
All estimators Accuracy 0.5065
First Estimator Accuracy 0.1875

2.

OneVsOne Classifier:
All estimators Estimator 0.5390
First Estimator Accuracy 0.3715

In [25]:
```
# Part 1 OneVsRest
from sklearn.metrics import accuracy_score
from sklearn.multiclass import OneVsRestClassifier
from sklearn.linear_model import LogisticRegression


#maybe do your EDA here?
foldnum = 0
lgr_fold_results = pd.DataFrame()

for train, test in cross_validation.KFold(len(gen_data), n_folds=10, shuf
    foldnum+=1

    [gen_tr_data, gen_te_data,
     gen_tr_target, gen_te_target] = folds_to_split(gen_data,gen_labels,t

    ovrc = OneVsRestClassifier(LogisticRegression(C=1e5, random_state=201
    ovrc.fit(gen_tr_data, gen_tr_target)

    lgr_fold_results.loc[foldnum, 'Default Accuracy'] = ovrc.score(gen_te

    firstEstimator = ovrc.estimators_[0]

    ovrc = OneVsRestClassifier(firstEstimator)
    ovrc.fit(gen_tr_data, gen_tr_target)

    # But a nicer way to store them is in a DataFrame
    lgr_fold_results.loc[foldnum, 'First Accuracy'] = metrics.accuracy_sc


print lgr_fold_results
print lgr_fold_results.mean()
```

```
    Default Accuracy  First Accuracy
1              0.535           0.170
2              0.490           0.140
3              0.475           0.170
4              0.470           0.230
5              0.465           0.190
6              0.530           0.200
7              0.555           0.210
8              0.545           0.170
9              0.525           0.205
10             0.475           0.190
Default Accuracy    0.5065
First Accuracy      0.1875
dtype: float64
```

In [26]:
```
# Part 2 OneVsOne

from sklearn.multiclass import OneVsOneClassifier

#maybe do your EDA here?
foldnum = 0
lgr_fold_results = pd.DataFrame()

for train, test in cross_validation.KFold(len(gen_data), n_folds=10, shuf
    foldnum+=1

    [gen_tr_data, gen_te_data,
     gen_tr_target, gen_te_target] = folds_to_split(gen_data,gen_labels,t

    ovoc = OneVsOneClassifier(LogisticRegression(C=1e5, random_state=2016
    ovoc.fit(gen_tr_data.values, gen_tr_target[0].values)
    lgr_fold_results.loc[foldnum, 'Default Est'] = ovoc.score(gen_te_data

    firstEstimator = ovoc.estimators_[0]

    ovoc = OneVsRestClassifier(firstEstimator)
    ovoc.fit(gen_tr_data, gen_tr_target)

    # But a nicer way to store them is in a DataFrame
    lgr_fold_results.loc[foldnum, 'First Est'] = metrics.accuracy_score(g

print len(ovoc.estimators_)
print lgr_fold_results
print lgr_fold_results.mean()
```

```
4
    Default Est  First Est
1         0.575      0.385
2         0.555      0.360
3         0.530      0.335
4         0.540      0.395
5         0.495      0.395
6         0.575      0.400
7         0.565      0.375
8         0.510      0.330
9         0.535      0.385
10        0.510      0.355
Default Est    0.5390
First Est      0.3715
dtype: float64
```

Question 4: Decision Boundaries (15 points)

At this point, you've seen most of the classifiers in scikit-learn (or learned about them in class). Let's try and get a broader sense of their similarities and differences. We'll do this by plotting the decision boundaries for each classifier in some simple datasets. This will mostly require you to copy code in the scikit-learn documentation, but you will need to understand it well enough to make some minor modifications

Replicate the Decision Boundaries sample (http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html) from scikit-learn, adding a fourth row for the Iris dataset for the Sepal Length and Petal Width attributes. Remove Linear Discriminant Analysis and Quadratic Discriminant Analysis from the list of methods, and instead add Logistic Regression and Perceptron. What does this plot tell you about different decision boundaries?

Answers 4:

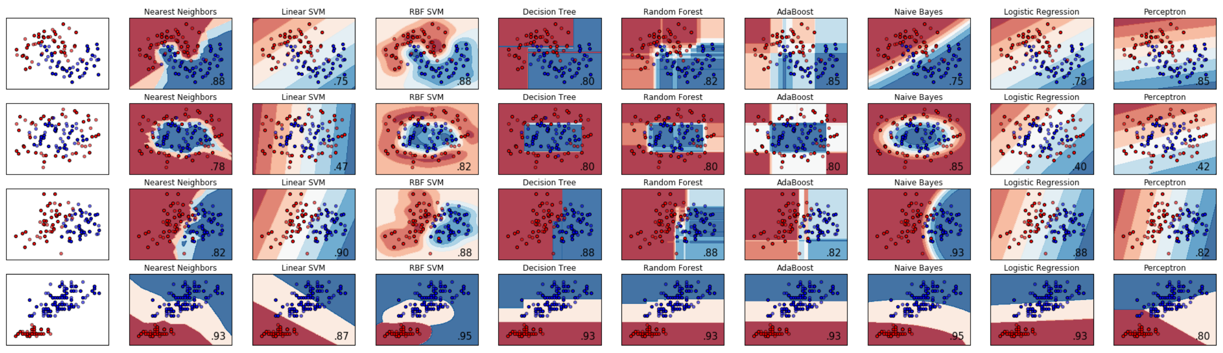
Observing the plots we can see several different things.
First we see that when the data is not linearly separatable, the Linear models have a very hard time fitting the data. In the make_circles dataset for example, we can see that the shape Linear SVM is nowhere close to fitting the data, and this can be seen from the low accuracy level of 47% compared to NN which is at 82% accuracy
Second, we can see that the decision tree model is able to achieve a better separation, but at the cost of complexity. The decision tree model is able to achieve this really nice, very square seaprating line, but the tree is very complex.

In [27]:

```
print(__doc__)


# Code source: Gaël Varoquaux
#              Andreas Müller
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.cross_validation import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classificatio
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.linear_model import Perceptron
from sklearn.linear_model import LogisticRegression

h = .02  # step size in the mesh

names = ["Nearest Neighbors", "Linear SVM", "RBF SVM", "Decision Tree",
         "Random Forest", "AdaBoost", "Naive Bayes", "Logistic Regression
         "Perceptron"]
classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    DecisionTreeClassifier(max_depth=5),
    RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
    AdaBoostClassifier(),
    GaussianNB(),
    LogisticRegression(C=1e5),
    Perceptron(n_iter = 5, random_state=20160202)]

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           random_state=1, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

iris = load_iris()

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable,
            iris
            ]
```

```
figure = plt.figure(figsize=(27, 9))
i = 1
# iterate over datasets
for ds in datasets:

    if (ds == iris):
        # We only take the two corresponding features
        X = ds.data[:, [0, 3]]
        y = ds.target
    else:
        # preprocess dataset, split into training and test part
        X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.

    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright)
    # and testing points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alph
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1

    # iterate over classifiers
    for name, clf in zip(names, classifiers):
        ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
        clf.fit(X_train, y_train)
        score = clf.score(X_test, y_test)

        # Plot the decision boundary. For that, we will assign a color to
        # point in the mesh [x_min, m_max]x[y_min, y_max].
        if hasattr(clf, "decision_function"):
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
            Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

        if (ds == iris):
            Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)
        # Plot also the training points
        ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_brigh
        # and testing points
        ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
                   alpha=0.6)

        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        ax.set_title(name)
        ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).lstrip('0'
                size=15, horizontalalignment='right')
        i += 1

figure.subplots_adjust(left=.02, right=.98)
plt.show()
```

```
Automatically created module for IPython interactive environment
```


Nearest Neighbors
Linear SVM
RBF SVM
Decision Tree
Random Forest
AdaBoost
Naive Bayes
Logistic Regression
Perceptron