Lab 15 - Support Vector Machines in Python

November 29, 2016

This lab on Support Vector Machines is a Python adaptation of p. 359-366 of "Introduction to Statistical Learning with Applications in R" by Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani. Original adaptation by J. Warmenhoven, updated by R. Jordan Crouser at Smith College for SDS293: Machine Learning (Spring 2016).

```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        from sklearn.metrics import confusion_matrix

        %matplotlib inline

        # We'll define a function to draw a nice plot of an SVM
        def plot_svc(svc, X, y, h=0.02, pad=0.25):
            x_min, x_max = X[:, 0].min()-pad, X[:, 0].max()+pad
            y_min, y_max = X[:, 1].min()-pad, X[:, 1].max()+pad
            xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_ma
            Z = svc.predict(np.c_[xx.ravel(), yy.ravel()])
            Z = Z.reshape(xx.shape)
            plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.2)

            plt.scatter(X[:,0], X[:,1], s=70, c=y, cmap=mpl.cm.Paired)
            # Support vectors indicated in plot by vertical lines
            sv = svc.support_vectors_
            plt.scatter(sv[:,0], sv[:,1], c='k', marker='x', s=100, linewidths='1')
            plt.xlim(x_min, x_max)
            plt.ylim(y_min, y_max)
            plt.xlabel('X1')
            plt.ylabel('X2')
            plt.show()
            print('Number of support vectors: ', svc.support_.size)
```

1 9.6 Lab: Support Vector Machines

In this lab, we'll use the `SVC` module from the `sklearn.svm` package to demonstrate the support vector classifier and the SVM:

```
In [2]: from sklearn.svm import SVC
```

2 9.6.1 Support Vector Classifier

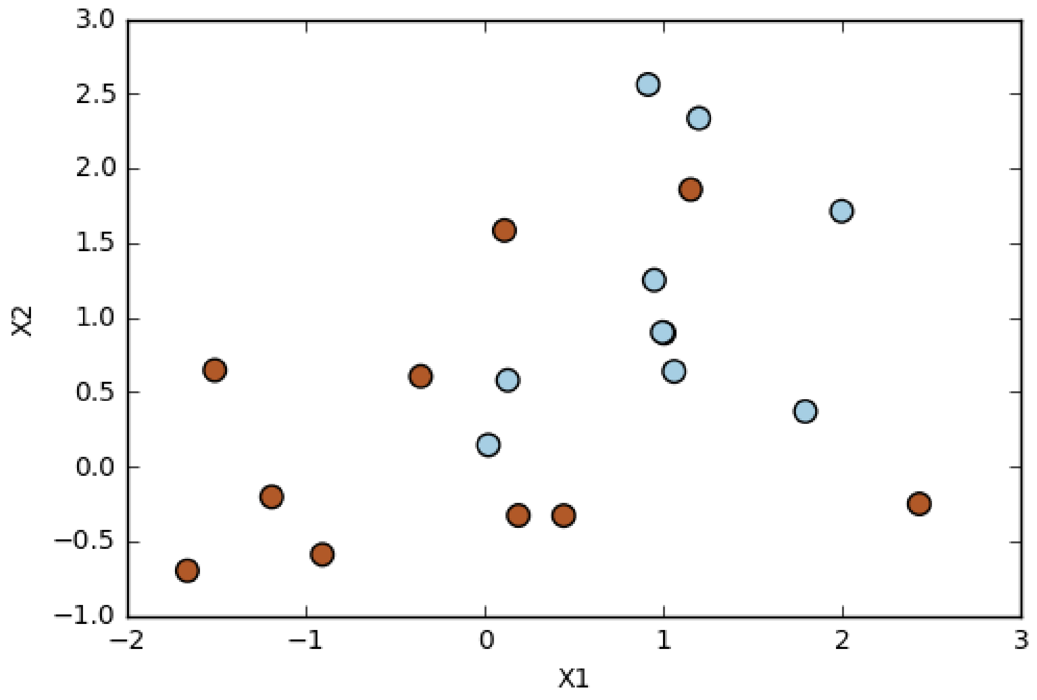
The `SVC()` function can be used to fit a support vector classifier when the argument `kernel` = ”`linear`” is used. This function uses a slightly different formulation of the equations we saw in lecture to build the support vector classifier. The `c` argument allows us to specify the cost of a violation to the margin. When the `c` argument is **small**, then the margins will be wide and many support vectors will be on the margin or will violate the margin. When the `c` argument is large, then the margins will be narrow and there will be few support vectors on the margin or violating the margin.

We can use the `SVC()` function to fit the support vector classifier for a given value of the `cost` parameter. Here we demonstrate the use of this function on a two-dimensional example so that we can plot the resulting decision boundary. Let’s start by generating a set of observations, which belong to two classes:

```
In [3]: # Generating random data: 20 observations of 2 features and divide into two
        np.random.seed(5)
        X = np.random.randn(20,2)
        y = np.repeat([1,-1], 10)

        X[y == -1] = X[y == -1] +1
```

Let’s plot the data to see whether the classes are linearly separable:

```
In [4]: plt.scatter(X[:,0], X[:,1], s=70, c=y, cmap=mpl.cm.Paired)
        plt.xlabel('X1')
        plt.ylabel('X2')

Out[4]: <matplotlib.text.Text at 0x117e01828>
```

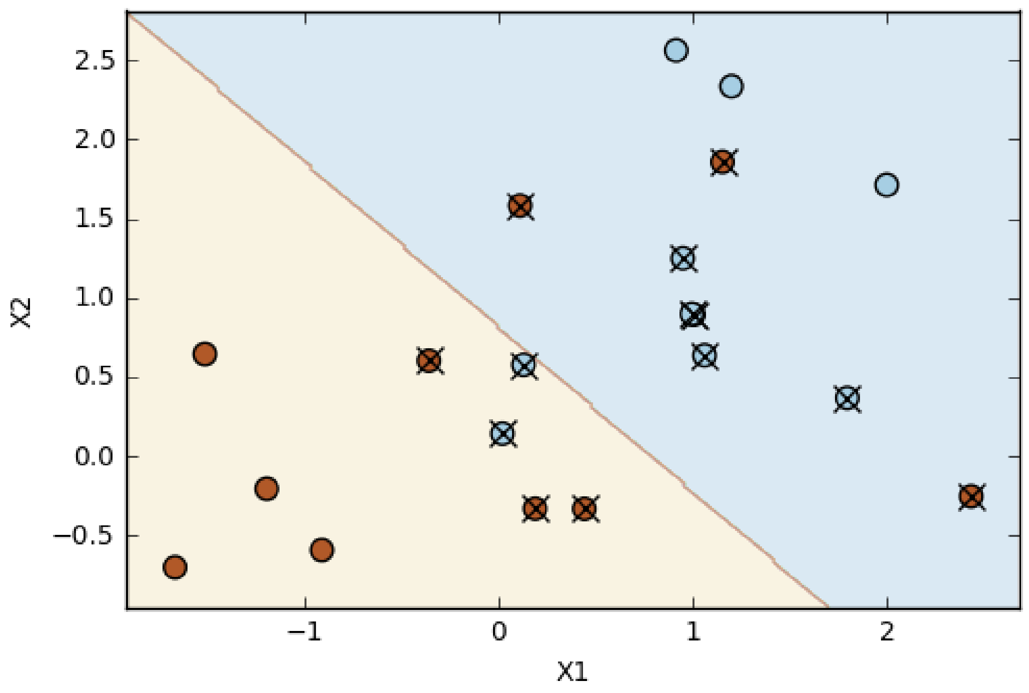


Nope; not linear. Next, we fit the support vector classifier:

```
In [5]: svc = SVC(C=1, kernel='linear')
        svc.fit(X, y)

Out[5]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
          decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
          max_iter=-1, probability=False, random_state=None, shrinking=True,
          tol=0.001, verbose=False)
```

We can now plot the support vector classifier by calling the `plot_svc()` function on the output of the call to `SVC()`, as well as the data used in the call to `SVC()`:

```
In [6]: plot_svc(svc, X, y)
```



```
Number of support vectors:  13
```

The region of feature space that will be assigned to the −1 class is shown in light blue, and the region that will be assigned to the +1 class is shown in brown. The decision boundary between the two classes is linear (because we used the argument `kernel` = "`linear`").

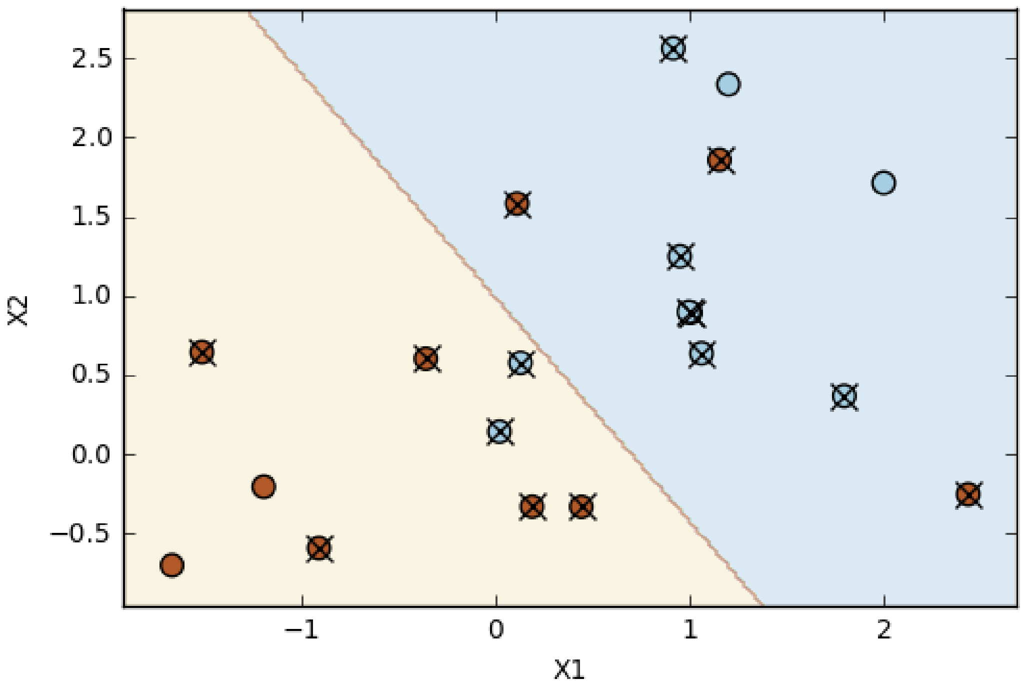
The support vectors are plotted with crosses and the remaining observations are plotted as circles; we see here that there are 13 support vectors. We can determine their identities as follows:

```
In [7]: svc.support_
```

```
Out[7]: array([10, 11, 13, 14, 15, 16, 17,  0,  1,  2,  4,  6,  8], dtype=int32)
```

What if we instead used a smaller value of the `cost` parameter?

```
In [8]: svc2 = SVC(C=0.1, kernel='linear')
        svc2.fit(X, y)
        plot_svc(svc2, X, y)
```



```
Number of support vectors:  16
```

Now that a smaller value of the c parameter is being used, we obtain a larger number of support vectors, because the margin is now **wider**.

The sklearn.grid_search module includes a a function GridSearchCV() to perform cross-validation. In order to use this function, we pass in relevant information about the set of models that are under consideration. The following command indicates that we want perform 10-fold cross-validation to compare SVMs with a linear kernel, using a range of values of the cost parameter:

```
In [9]: from sklearn.grid_search import GridSearchCV

        # Select the optimal C parameter by cross-validation
        tuned_parameters = [{'C': [0.001, 0.01, 0.1, 1, 5, 10, 100]}]
        clf = GridSearchCV(SVC(kernel='linear'), tuned_parameters, cv=10, scoring='
        clf.fit(X, y)

/Users/jcrouser/anaconda3/lib/python3.5/site-packages/sklearn/cross_validation.py:4
  "This module will be removed in 0.20.", DeprecationWarning)
/Users/jcrouser/anaconda3/lib/python3.5/site-packages/sklearn/grid_search.py:43: De
  DeprecationWarning)


Out[9]: GridSearchCV(cv=10, error_score='raise',
               estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
          decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
          max_iter=-1, probability=False, random_state=None, shrinking=True,
          tol=0.001, verbose=False),
               fit_params={}, iid=True, n_jobs=1,
               param_grid=[{'C': [0.001, 0.01, 0.1, 1, 5, 10, 100]}],
               pre_dispatch='2*n_jobs', refit=True, scoring='accuracy', verbose=0)
```

We can easily access the cross-validation errors for each of these models:

```
In [10]: clf.grid_scores_
```

```
Out[10]: [mean: 0.80000, std: 0.24495, params: {'C': 0.001},
          mean: 0.80000, std: 0.24495, params: {'C': 0.01},
          mean: 0.80000, std: 0.24495, params: {'C': 0.1},
          mean: 0.75000, std: 0.33541, params: {'C': 1},
          mean: 0.75000, std: 0.33541, params: {'C': 5},
          mean: 0.75000, std: 0.33541, params: {'C': 10},
          mean: 0.75000, std: 0.33541, params: {'C': 100}]
```

The `GridSearchCV`() function stores the best parameters obtained, which can be accessed as follows:

```
In [11]: clf.best_params_
```

```
Out[11]: {'C': 0.001}
```

c=0.001 is best according to `GridSearchCV`.

As usual, the `predict`() function can be used to predict the class label on a set of test observations, at any given value of the cost parameter. Let's generate a test data set:

```
In [12]: np.random.seed(1)
         X_test = np.random.randn(20,2)
         y_test = np.random.choice([-1,1], 20)
         X_test[y_test == 1] = X_test[y_test == 1] -1
```

Now we predict the class labels of these test observations. Here we use the best model obtained through cross-validation in order to make predictions:

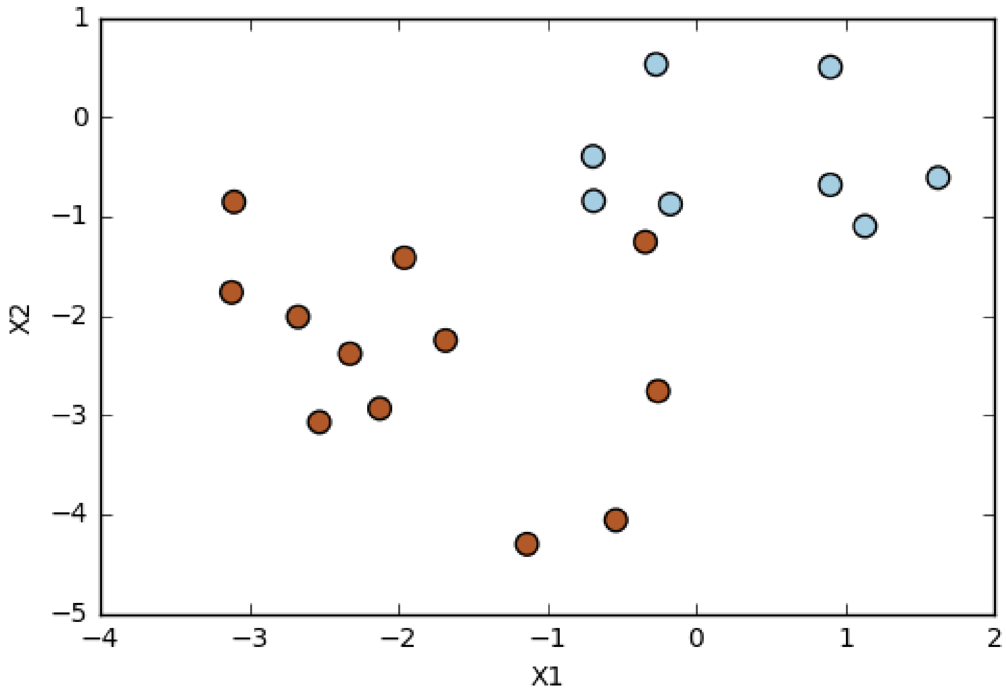
```
In [13]: svc2 = SVC(C=0.001, kernel='linear')
         svc2.fit(X, y)
         y_pred = svc2.predict(X_test)
         pd.DataFrame(confusion_matrix(y_test, y_pred), index=svc2.classes_, columm
```

```
Out[13]:     -1   1
         -1   2   6
          1   0  12
```

With this value of `c`, 14 of the test observations are correctly classified.

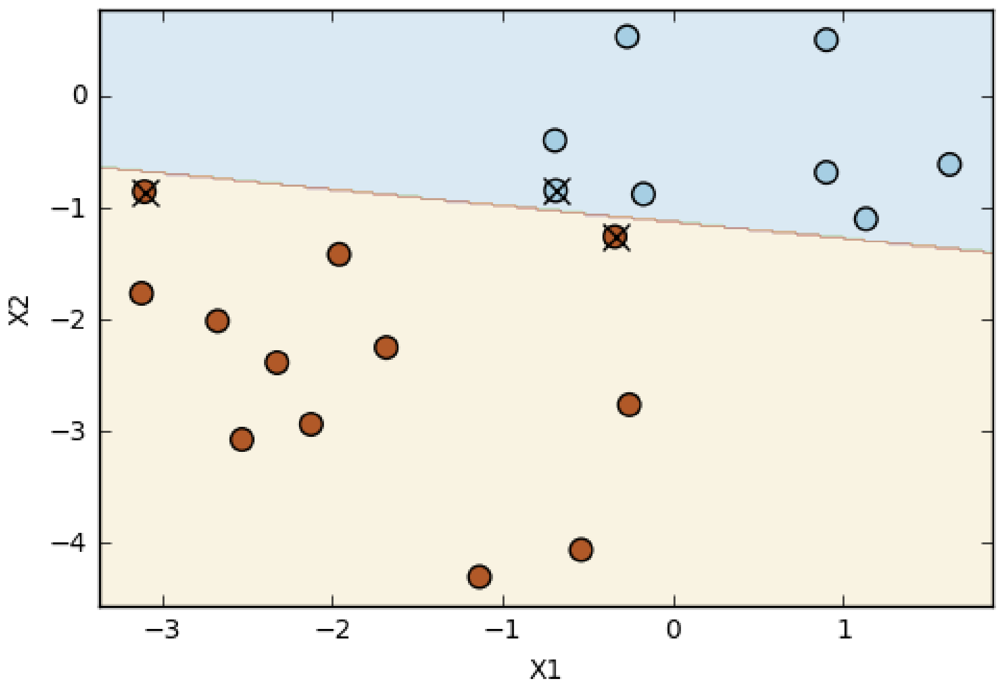
Now consider a situation in which the two classes are linearly separable. Then we can find a separating hyperplane using the `svm`() function. First we'll give our simulated data a little nudge so that they are linearly separable:

```
In [14]: X_test[y_test == 1] = X_test[y_test == 1] -1
         plt.scatter(X_test[:,0], X_test[:,1], s=70, c=y_test, cmap=mpl.cm.Paired)
         plt.xlabel('X1')
         plt.ylabel('X2')

Out[14]: <matplotlib.text.Text at 0x11845b358>
```



Now the observations are **just barely linearly** separable. We fit the support vector classifier and plot the resulting hyperplane, using a very large value of `cost` so that no observations are misclassified.

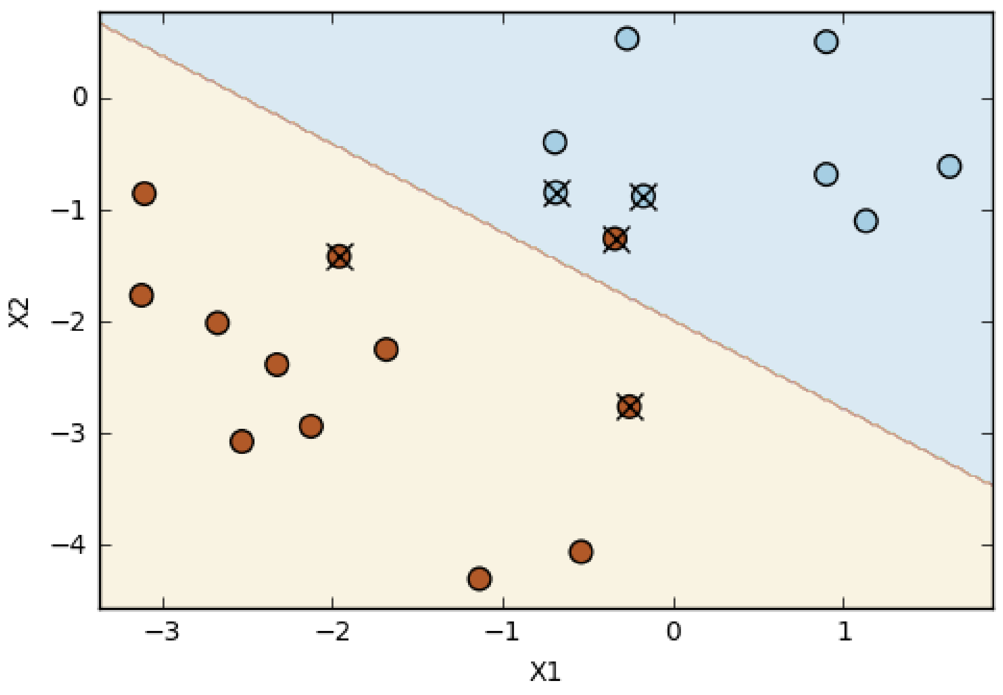
```
In [15]: svc3 = SVC(C=1e5, kernel='linear')
         svc3.fit(X_test, y_test)
         plot_svc(svc3, X_test, y_test)
```



```
Number of support vectors:  3
```

No training errors were made and only three support vectors were used. However, we can see from the figure that the margin is very narrow (because the observations that are **not** support vectors, indicated as circles, are very close to the decision boundary). It seems likely that this model will perform poorly on test data. Let's try a smaller value of `cost`:

```
In [16]: svc4 = SVC(C=1, kernel='linear')
         svc4.fit(X_test, y_test)
         plot_svc(svc4, X_test, y_test)
```



```
Number of support vectors:  5
```

Using `cost` = `1`, we misclassify a training observation, but we also obtain a much wider margin and make use of five support vectors. It seems likely that this model will perform better on test data than the model with `cost` = `1e5`.

3 9.6.2 Support Vector Machine

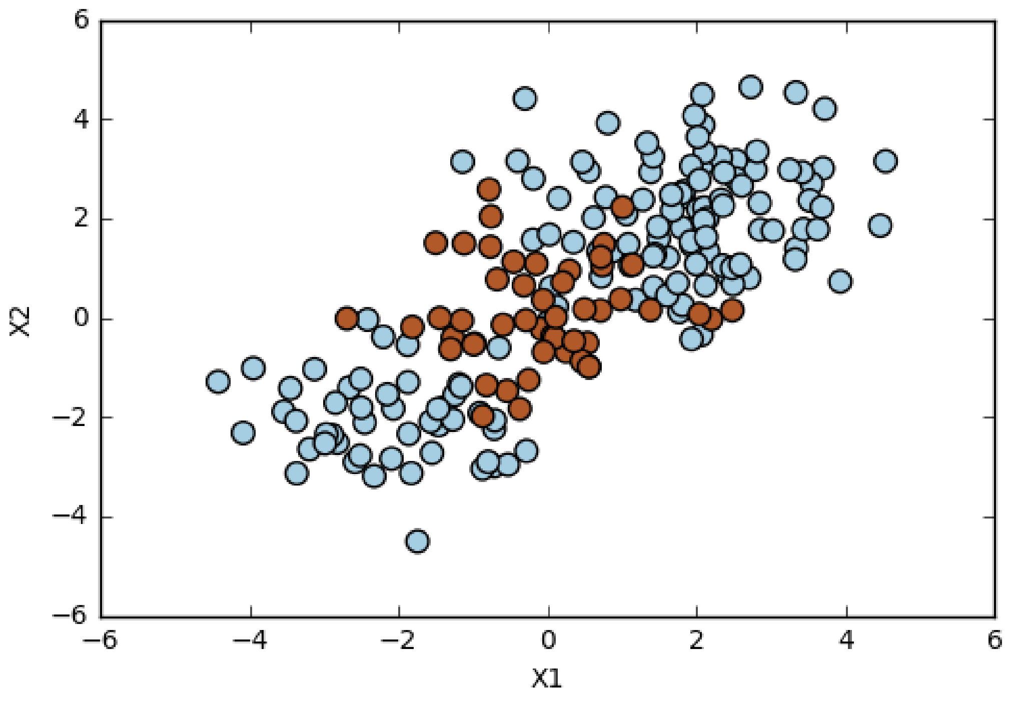
In order to fit an SVM using a **non-linear kernel**, we once again use the `SVC()` function. However, now we use a different value of the parameter kernel. To fit an SVM with a polynomial kernel we use `kernel` = ”`poly`”, and to fit an SVM with a radial kernel we use `kernel` = ”`rbf`”. In the former case we also use the `degree` argument to specify a degree for the polynomial kernel, and in the latter case we use `gamma` to specify a value of γ for the radial basis kernel.

Let's generate some data with a non-linear class boundary:

```
In [20]: from sklearn.model_selection import train_test_split

         np.random.seed(8)
         X = np.random.randn(200,2)
         X[:100] = X[:100] +2
         X[101:150] = X[101:150] -2
         y = np.concatenate([np.repeat(-1, 150), np.repeat(1,50)])
```

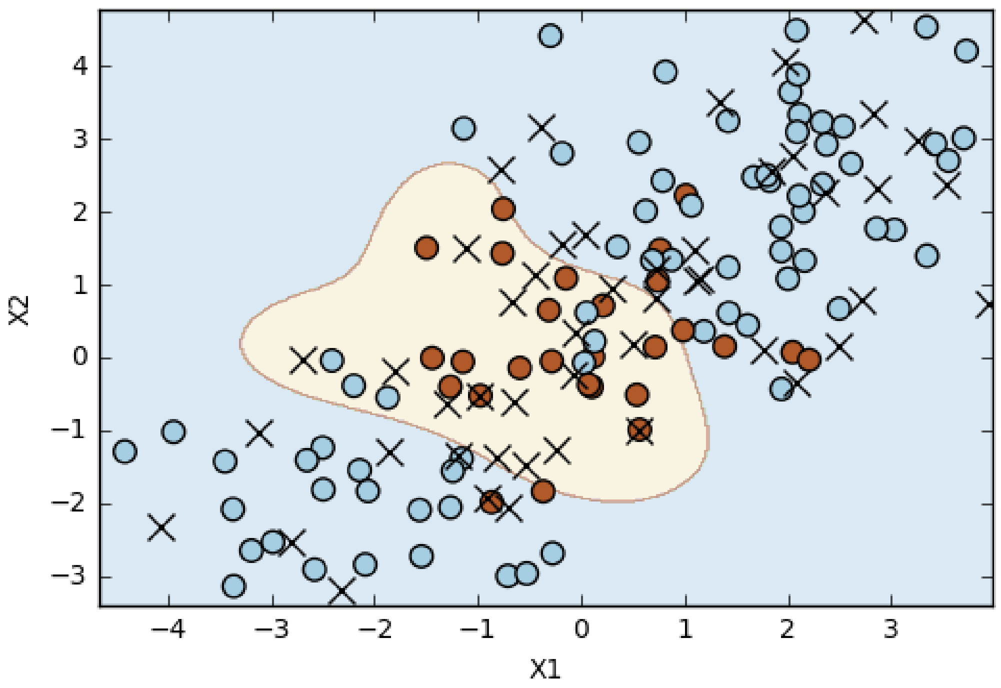
```
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.5,

plt.scatter(X[:,0], X[:,1], s=70, c=y, cmap=mpl.cm.Paired)
plt.xlabel('X1')
plt.ylabel('X2')
```

```
Out[20]: <matplotlib.text.Text at 0x1190d3e10>
```



See how one class is kind of stuck in the middle of another class? This suggests that we might want to use a **radial kernel** in our SVM. Now let's fit the training data using the `SVC()` function with a radial kernel and $\gamma = 1$:

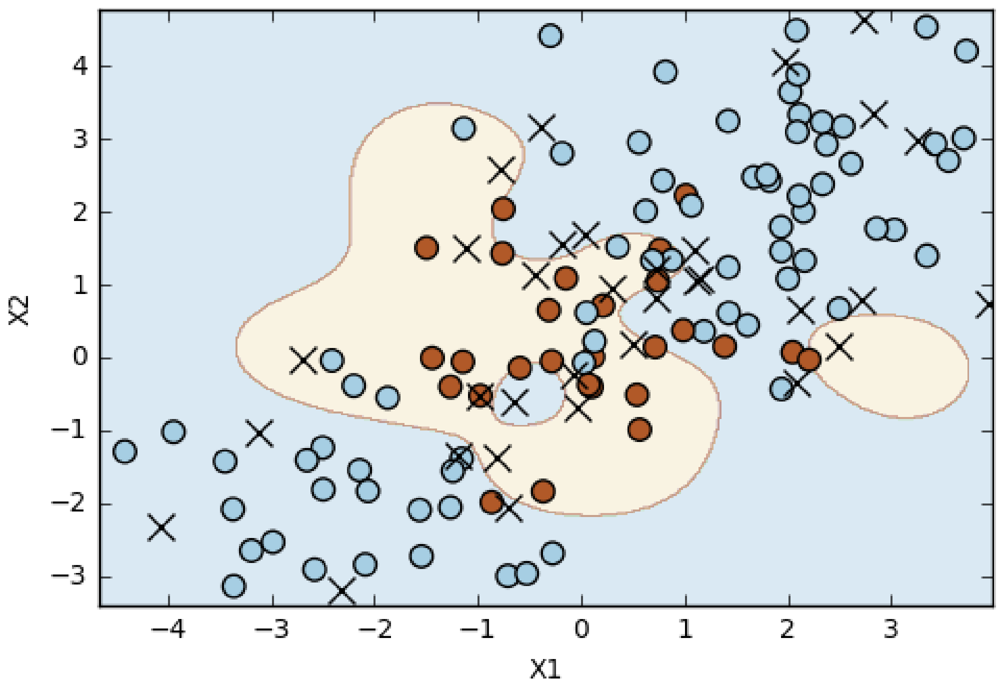
```
In [21]: svm = SVC(C=1.0, kernel='rbf', gamma=1)
         svm.fit(X_train, y_train)
         plot_svc(svm, X_test, y_test)
```



```
Number of support vectors:  51
```

Not too shabby! The plot shows that the resulting SVM has a decidedly non-linear boundary. We can see from the figure that there are a fair number of training errors in this SVM fit. If we increase the value of cost, we can reduce the number of training errors:

```
In [22]: # Increasing C parameter, allowing more flexibility
         svm2 = SVC(C=100, kernel='rbf', gamma=1.0)
         svm2.fit(X_train, y_train)
         plot_svc(svm2, X_test, y_test)
```



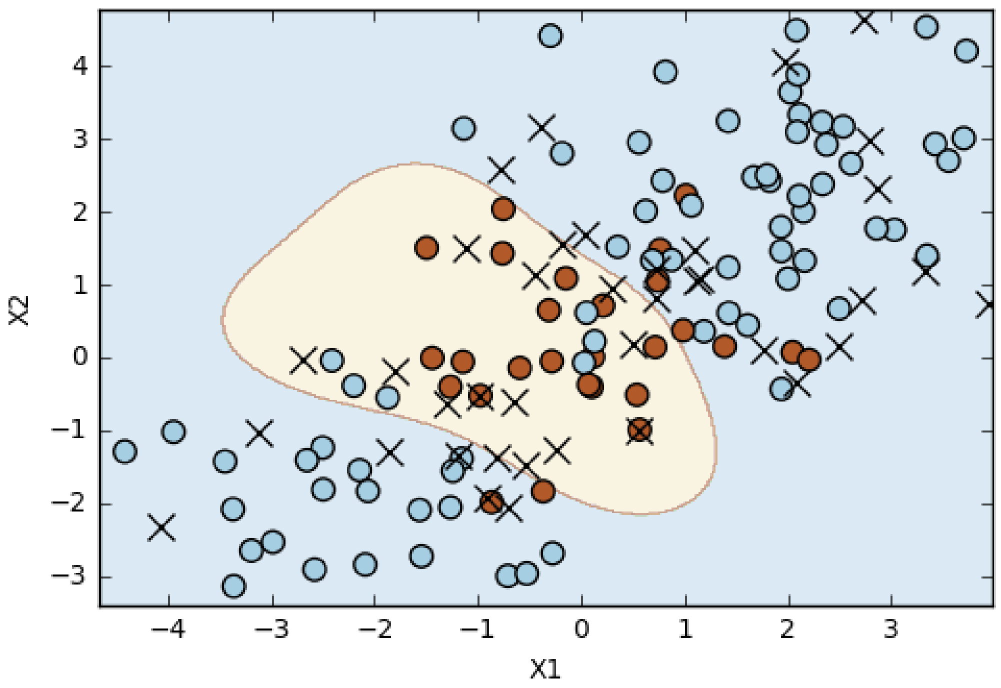
```
Number of support vectors:  36
```

However, this comes at the price of a more irregular decision boundary that seems to be at risk of overfitting the data. We can perform cross-validation using `GridSearchCV()` to select the best choice of γ and cost for an SVM with a radial kernel:

```
In [23]: tuned_parameters = [{'C': [0.01, 0.1, 1, 10, 100],
                              'gamma': [0.5, 1,2,3,4]}]
         clf = GridSearchCV(SVC(kernel='rbf'), tuned_parameters, cv=10, scoring='ac
         clf.fit(X_train, y_train)
         clf.best_params_

Out[23]: {'C': 1, 'gamma': 0.5}
```

Therefore, the best choice of parameters involves `cost` = 1 and `gamma` = 0.5. We can plot the resulting fit using the `plot_svc()` function, and view the test set predictions for this model by applying the `predict()` function to the test data:

```
In [24]: plot_svc(clf.best_estimator_, X_test, y_test)
         print(confusion_matrix(y_test, clf.best_estimator_.predict(X_test)))
         print(clf.best_estimator_.score(X_test, y_test))
```



```
Number of support vectors:  41
[[67  6]
 [ 9 18]]
0.85
```

85% of test observations are correctly classified by this SVM. Not bad!

4 9.6.3 ROC Curves

The auc() function from the sklearn.metrics package can be used to produce ROC curves such as those we saw in lecture:

```
In [25]: from sklearn.metrics import auc
         from sklearn.metrics import roc_curve
```

Let's start by fitting two models, one more flexible than the other:

```
In [26]: # More constrained model
         svm3 = SVC(C=1, kernel='rbf', gamma=1)
         svm3.fit(X_train, y_train)
```

```
Out[26]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape=None, degree=3, gamma=1, kernel='rbf',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False)
```

```
In [27]: # More flexible model
         svm4 = SVC(C=1, kernel='rbf', gamma=50)
         svm4.fit(X_train, y_train)

Out[27]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape=None, degree=3, gamma=50, kernel='rbf',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False)
```

SVMs and support vector classifiers output class labels for each observation. However, it is also possible to obtain fitted values for each observation, which are the numerical scores used to obtain the class labels. For instance, in the case of a support vector classifier, the fitted value for an observation $X = (X_1, X_2, ..., X_p)^T$ takes the form $\hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2 + ... + \hat{\beta}_p X_p$.

For an SVM with a non-linear kernel, the equation that yields the fitted value is given in (9.23) on p. 352 of the ISLR book. In essence, the sign of the fitted value determines on which side of the decision boundary the observation lies. Therefore, the relationship between the fitted value and the class prediction for a given observation is simple: if the fitted value exceeds zero then the observation is assigned to one class, and if it is less than zero than it is assigned to the other.

In order to obtain the fitted values for a given SVM model fit, we use the `.decision_function()` method of the SVC:

```
In [ ]: y_train_score3 = svm3.decision_function(X_train)
        y_train_score4 = svm4.decision_function(X_train)
```

Now we can produce the ROC plot to see how the models perform on both the training and the test data:

```
In [ ]: y_train_score3 = svm3.decision_function(X_train)
        y_train_score4 = svm4.decision_function(X_train)

        false_pos_rate3, true_pos_rate3, _ = roc_curve(y_train, y_train_score3)
        roc_auc3 = auc(false_pos_rate3, true_pos_rate3)

        false_pos_rate4, true_pos_rate4, _ = roc_curve(y_train, y_train_score4)
        roc_auc4 = auc(false_pos_rate4, true_pos_rate4)

        fig, (ax1,ax2) = plt.subplots(1, 2, figsize=(14,6))
        ax1.plot(false_pos_rate3, true_pos_rate3, label='SVM $\gamma = 1$ ROC curve
        ax1.plot(false_pos_rate4, true_pos_rate4, label='SVM $\gamma = 50$ ROC curv
        ax1.set_title('Training Data')

        y_test_score3 = svm3.decision_function(X_test)
        y_test_score4 = svm4.decision_function(X_test)

        false_pos_rate3, true_pos_rate3, _ = roc_curve(y_test, y_test_score3)
        roc_auc3 = auc(false_pos_rate3, true_pos_rate3)

        false_pos_rate4, true_pos_rate4, _ = roc_curve(y_test, y_test_score4)
        roc_auc4 = auc(false_pos_rate4, true_pos_rate4)
        ax2.plot(false_pos_rate3, true_pos_rate3, label='SVM $\gamma = 1$ ROC curve
        ax2.plot(false_pos_rate4, true_pos_rate4, label='SVM $\gamma = 50$ ROC curv
        ax2.set_title('Test Data')

        for ax in fig.axes:
            ax.plot([0, 1], [0, 1], 'k--')
            ax.set_xlim([-0.05, 1.0])
            ax.set_ylim([0.0, 1.05])
            ax.set_xlabel('False Positive Rate')
            ax.set_ylabel('True Positive Rate')
            ax.legend(loc="lower right")
```

To get credit for this lab, describe what the ROC plot is telling you about the SVM's performance on the test data and post to Piazza: https://piazza.com/class/igwiv4w3ctb6rg?cid=54