CS416 – HW3

1 Kernel (10 points)

Suppose $x, z \in \mathbb{R}^d$, and let's consider the following function

$$K(x, z) = (x^T z)^2$$

Let's prove $K(x, z)$ is the kernel function corresponding to the feature mapping ϕ give by

$$\phi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ \cdots \\ x_1 x_d \\ x_2 x_1 \\ x_2 x_2 \\ \cdots \\ x_2 x_d \\ \cdots \\ x_d x_1 \\ \cdots \\ x_d x_d \end{bmatrix}$$

The proof is very simple. We merely check that $K(x, z) =< \phi(x), \phi(z) >$

$$K(x, z) = (\sum_{i=1}^{d} x_i z_i)(\sum_{i=1}^{d} x_j z_j) = \sum_{i=1}^{d}\sum_{j=1}^{d} x_i x_j z_i z_j = \sum_{i,j=1}^{d} (x_i x_j)(z_i z_j) =< \phi(x), \phi(z) >$$

Please show what feature mapping the following kernel function corresponds to and prove the kernel function corresponds to the feature mapping. Show the running time of computing the kernel function of two vectors and that of computing the inner product of two vectors.

$$K(x, z) = (x^T z + 1)^3$$

2 SVM (5 points + 10 points)

1. Suppose you are given the following training data

positive: (1,2,3) (1,1,4)
negative: (3,2,-1) (4,3,-2) (3,5,-3)

Write down the SVM optimization for those training data including the optimization objective and the constraints.

2. Suppose you are given the following training data

positive: (1,2) (1,1) (2,1) (0,1)
negative: (3,2) (4,3) (3,5)

Which points are support vectors? What is the decision boundary if you use SVM? In this problem, you can simply look at the points and decide which points are support vectors and then calculate the decision boundary.

3 Decision Tree (20 points)

Consider the following dataset consisting of five training examples followed by three test examples:

x1	x2	x3	y
-	+	+	-
+	+	+	+
-	+	-	+
-	-	+	-
+	+	-	+
+	-	-	?
-	-	-	?
+	-	+	?

There are three attributes (or features or dimensions), x1, x2 and x3, taking the values + and -. The label (or class) is given in the last column denoted y; it also takes the two values + and -.

Simulate each of the following learning algorithms on this dataset. In each case, show the final hypothesis that is induced, and show how it was computed. Also, say what its prediction would be on the three test examples.

- The decision tree algorithm discussed in class. For this algorithm, use the information gain (entropy) impurity measure as a criterion for choosing an attribute to split on. Grow your tree until all nodes are pure, but do not attempt to prune the tree.

- AdaBoost. For this algorithm, you should interpret label values of + and - as the real numbers +1 and -1. Use decision stumps as weak hypotheses, and assume that the weak learner always computes the decision stump with minimum error on the training set weighted in AdaBoost algorithm. Note that a decision stump is a one-level decision tree. Run your boosting algorithm for three rounds and list the intermediate results.

Programming: Support Vector Machines (20 points)

In this section, we'll implement various kernels for the support vector machine (SVM). This exercise looks long, but in practice you'll be writing only a few lines of code. The scikit learn package already includes several SVM implementations; in this case, we'll focus on the SVM for classification, sklearn.svm.SVC. Before starting this assignment, be certain to read through the documentation for SVC, available at

`http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`

While we could certainly create our own SVM implementation, most people applying SVMs to real problems rely on highly optimized SVM toolboxes, such as LIBSVM or SVMlight[1]. These toolboxes provide highly optimized SVM implementations that use a variety of optimization techniques to enable them to scale to extremely large problems. Therefore, we will focus on implementing custom kernels for the SVMs, which is often essential for applying these SVM toolboxes to real applications.

Relevant Files in this homework Skeleton[2]

`example_svm.py`
`example_svmCustomKernel.py`
`* svmKernels.py`
`* test_svm_parameters.ipynb`
`test_svmPolyKernel.py`
`data/svmData.dat`
`data/svmTuningData.dat`
`test_svmGaussianKernel.py`

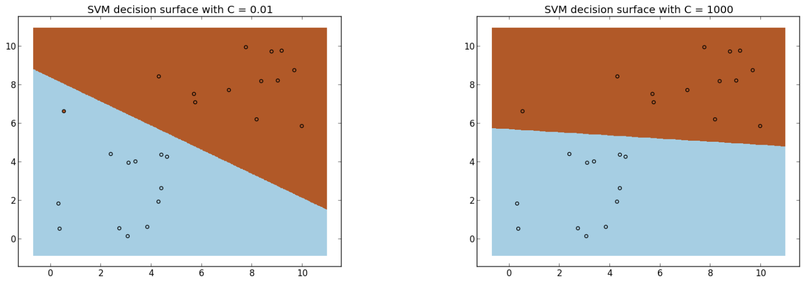
1.1 Getting Started

The SVC implementation provided with scikit learn uses the parameter C to control the penalty for misclassifying training instances. We can think of C as being similar to the inverse of the regularization parameter $\frac{1}{\lambda}$ that we used before for linear and logistic regression. C = 0 causes the SVM to incur no penalty for misclassifications, which will encourage it to fit a larger-margin hyperplane, even if that hyperplane misclassifies more training instances. As C grows large, it causes the SVM to try to classify all training examples correctly, and so it will choose a smaller margin hyperplane if that hyperplane fits the training data better.

Examine example_svm.py, which fits a linear SVM to the data shown below. Note that most of the positive and negative instances are grouped together, suggesting a clear separation between the classes, but there is an outlier around (0.5,6.2). In the first part of this exercise, we will see how this outlier affects the SVM fit.

[1] The SVC implementation provided with scikit_learn is based on LIBSVM, but is not quite as efficient.

[2] `*` indicates files that you will need to complete; you should not need to modify any of the other files.



(a) A

(b) B

Figure 1:

Run example svm.py with C = 0.01, and you can clearly see that the hyperplane follows the natural separation between most of the data, but misclassifies the outlier. Try increasing the value of C and observe the effect on the resulting hyperplane. With C = 1,000, we can see that the decision boundary correctly classifies all training data, but clearly no longer captures the natural separation between the data.

1.2 Implementing Custom Kernels

The SVC implementation allows us to define our own kernel functions to learn non-linear decision surfaces using the SVM. The SVC constructor's kernel argument can be defined as either a string specifying one of the built-in kernels (e.g., 'linear', 'poly' (polynomial), 'rbf' (radial basis function), 'sigmoid', etc.) or it can take as input a custom kernel function, as we will define in this exercise. For example, the following code snippet defines a custom kernel and uses it to train the SVM:

```
def myCustomKernel (X1 , X2 ) :
    """
    Custom kernel :
    k(X1, X2) = X1 (3 0) X2.T
                    (0 2)
    Note that X1 and X2 are numpy arrays,
    so we must use .dot to multiply them.
    """
    M = np.array([[3.0, 0], [0, 2.0]])
    return np.dot(np.dot(X1, M), X2.T)

# create SVM with custom kernel and train model
clf = svm.SVC(kernel=myCustomKernel )
clf.fit(X, Y)
```

When the SVM calls the custom kernel function during training, X1 and X2 are both initialized to be the same as X (i.e., $n_{train} - by - d$ numpy arrays); in other words, they both contain a complete copy of the training instances. The custom kernel function returns an $n_{train} - by - n_{train}$ numpy array during the training step. Later, when it is used for testing, X1 will be the n_{test} testing instances and X2 will be the n_{train} training instances, and so it will return an $n_{test} - by - n_{train}$ numpy array. For a complete example, see **example_svmCustomKernel.py**, which uses the custom kernel above to generate the following figure:

1.3 Implementing the Polynomial Kernel (5 points)

We will start by writing our own implementation of the polynomial kernel and incorporate it into the SVM.[3] Complete the myPolynomialKernel() function in svmKernels.py to implement the polynomial kernel:

$$K(v, w) = (< v, w > +1)^d$$

where d is the degree of polynomial and the "+1" incorporates all lower-order polynomial terms. In this case, v and w are feature vectors. Vectorize your implementation to make it fast. Once complete, run test_svmPolyKernel.py to produce a plot of the decision surface. For comparison, it also shows the decision surface learned using the equivalent built-in polynomial kernel; your results should be identical.

For the built-in polynomial kernel, the degree is specified in the SVC constructor. However, in our custom kernel we must set the degree via a global variable _polyDegree. Therefore, be sure to use the value of the global variable polyDegree as the degree in your polynomial kernel. The test_svmPolyKernel.py script uses _polyDegree for the degree of both your custom kernel and the built-in polynomial kernel. Vary both C and d and study how the SVM reacts.

[3] Although scikit learn already defines the polynomial kernel, defining our own version of it provides an easy way to get started implementing custom kernels.

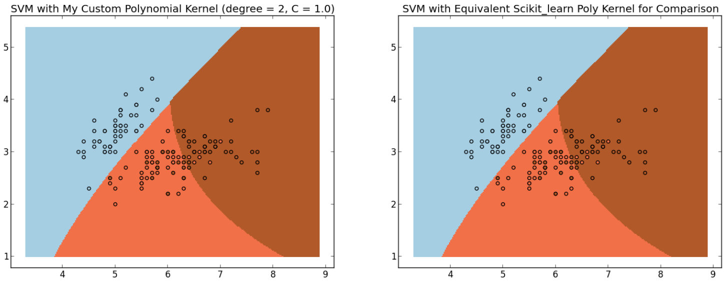
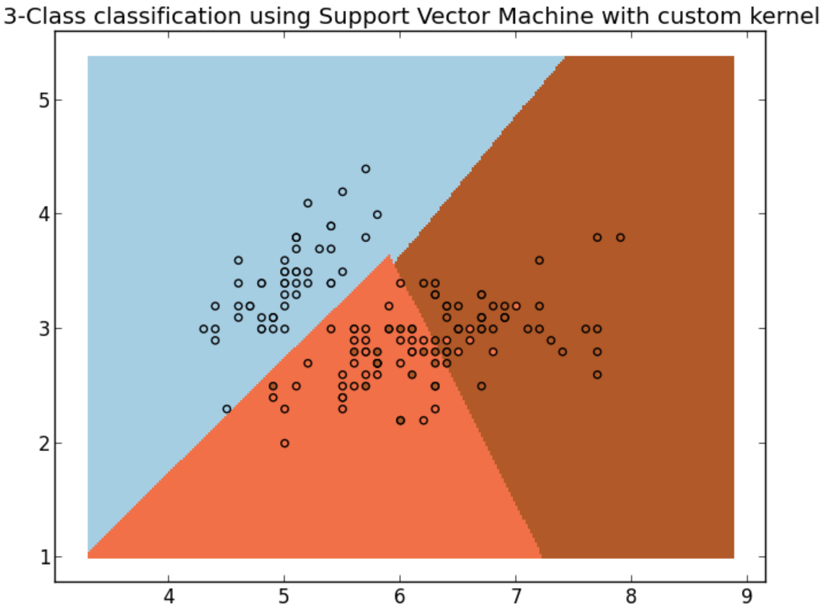


Figure 2: Sample output of test_svmPolyKernel.py



1.4 Implementing the Gaussian Radial Basis Function Kernel (5 points)

Next, complete the myGaussianKernel() function in svmKernels.py to implement the Gaussian kernel:

$$K(v, w) = exp(-\frac{||v - w||_2^2}{2\sigma^2})$$

Be sure to use the **_gaussSigma** for σ in your implementation. For computing the pairwise squared distances between the points, you must write the method to compute it yourself; specifically you may not use the helper methods available

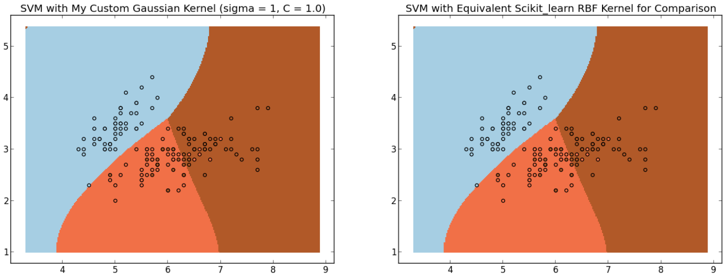


Figure 3: Sample output of test_svmGaussianKernel.py

in **sklearn.metrics.pairwise** or that come with scipy. You can test your implementation and compare it to the equivalent RBF-kernel provided in sklearn by running test **svmGaussianKernel.py**.

Again, vary both C and σ and study how the SVM reacts.

Write a brief paragraph describing how the SVM reacts as both C and d vary for the polynomial kernel, and as C and σ vary for the Gaussian kernel. Put this paragraph in your writeup.

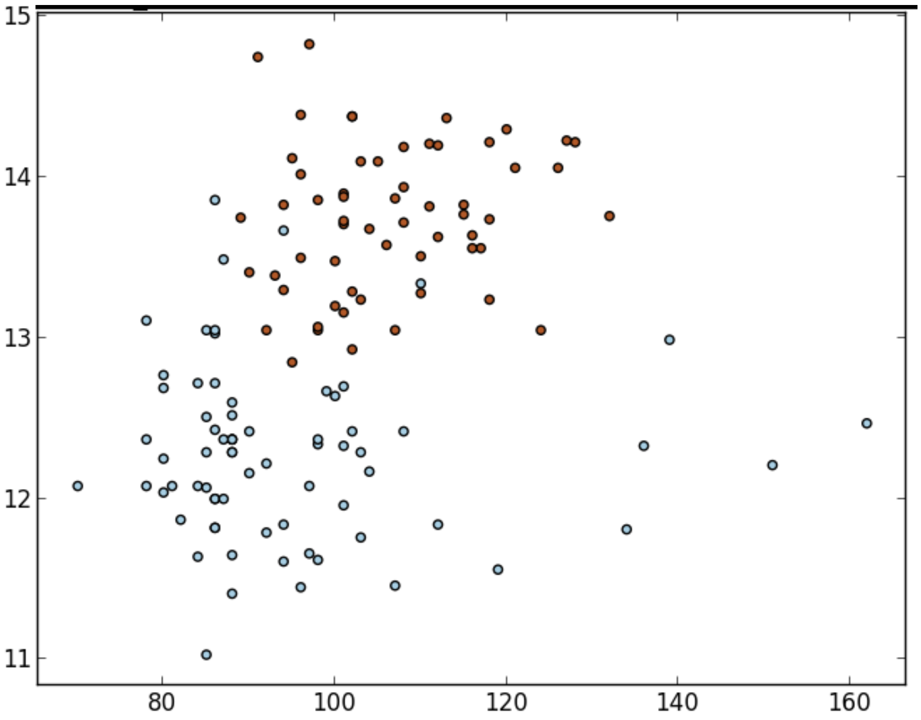
1.5 Choosing the Optimal Parameters (10 points)

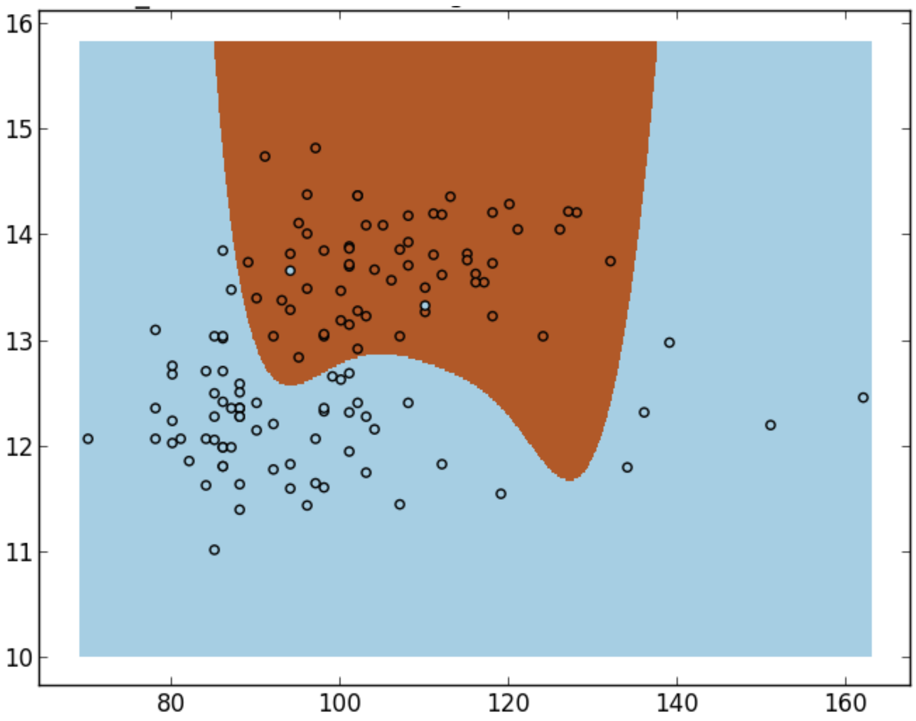
This exercise will further help you gain further practical skill in applying SVMs with Gaussian kernels. Choosing the correct values for C and σ can dramatically affect the quality of the model's fit to data. Your task is to determine the optimal values of C and σ for an SVM with your Gaussian kernel as applied to the data in data/svmTuningData.dat, depicted below. You should search over the space of possible combinations of C and σ, and determine the optimal fit as measured by accuracy. The file for this exercise is test_svm_parameters.ipynb We recommend that you search over multiplicative steps (e.g., ..., 0.01, 0.03, 0.06, 0.1, 0.3, 0.6, 1, 3, 6, 10, 30, 60, 100, ...). Once you determine the optimal parameter values, report those optimal values and the corresponding estimated accuracy in your test_svm_parameters.ipynb. For reference, the SVM with the optimal parameters we found produced the following decision surface.

The resulting decision surface for your optimal parameters may look slightly different than ours.

To have a good estimate of your model's performance, you should average the accuracy over a certain number trials of 10-fold cross-validation over the data set. Make certain to observe the following details:

- For each trial, split the data randomly into 10 folds, choose one to be the "test" fold and train the decision tree classifier on the remaining nine folds. Then, evaluate the trained model on the held-out "test" fold to obtain its performance.


15
14
13
12
11
80
100
120
140
160


16
15
14
13
12
11
10
80
100
120
140
160

- Repeat this process until each fold has been the test fold exactly once, then advance to the next trial.
- Be certain to shuffle the data at the start of each trial, but never within a trial. Report the mean of the prediction accuracy over all trials of 10-fold cross validation.

Note: although scikit-learn provides libraries that implement cross-fold validation, you may not use them for this assignment – you must implement cross-fold validation yourself.