MACHINE LEARNING

Machine Learning Exercises In Python, Part 6

26TH JUNE 2016

*This post is part of a series covering the exercises from Andrew Ng's **machine learning** class on Coursera. The original code, exercise text, and data files for this post are available **here**.*

We're now hitting the home stretch of both the course content and this series of blog posts. In this exercise, we'll be using support vector machines (SVMs) to build a spam classifier. We'll start with SVMs on some simple 2D data sets to see how they work. Then we'll look at a set of email data and build a classifier on the processed emails using a SVM to determine if they are spam or not. As always, it helps to follow along using the exercise text for the course (posted **here**).

Before jumping into the code, let's quickly recap what an SVM is and why it's worth learning about. Broadly speaking, SVMs are a class of supervised learning algorithm that builds a representation of the examples in the training data as points in space, mapped so that the examples belonging to each class in the data are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a class based on which side of the gap they fall on. SVMs are a binary classifcation tool by default, although there are ways to use them in multi-class scenarios. SVMs can also handle non-linear classification using something called the **kernel trick** to project the data into a high-dimensional space before attempting to find a hyperplane. SVMs are a powerful class of algorithms and are used often in practical machine learning applications.

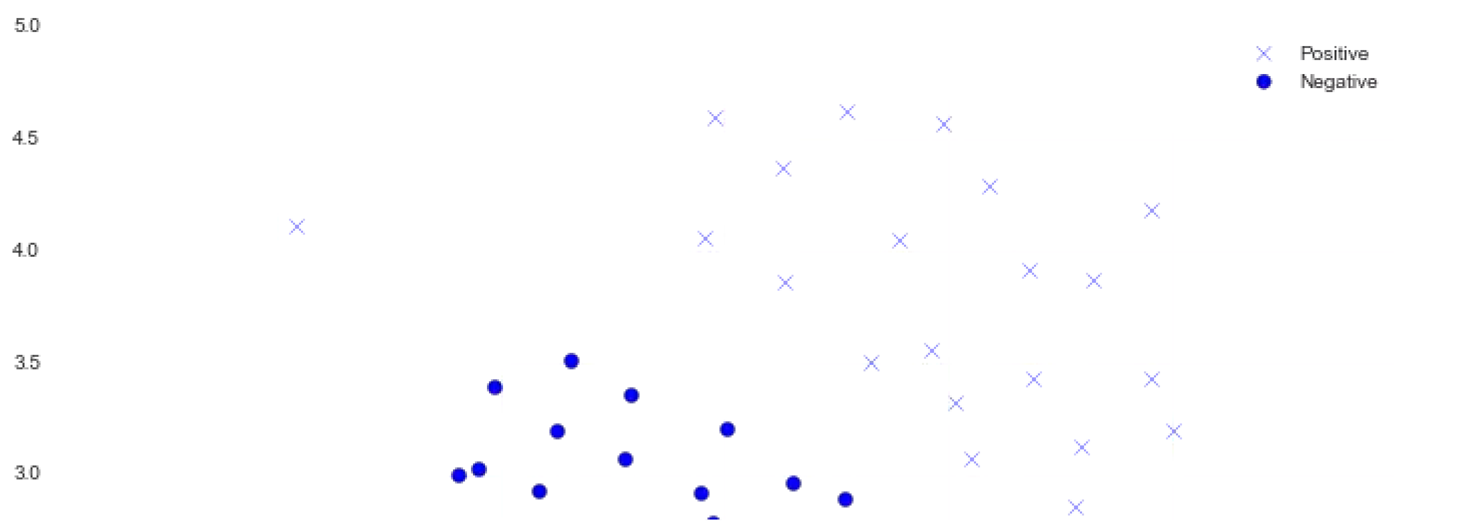
The first thing we're going to do is look at a simple 2-dimensional data set and see how a linear SVM works on the data set for varying values of C (similar to the regularization term in linear/logistic regression). Let's load the data.

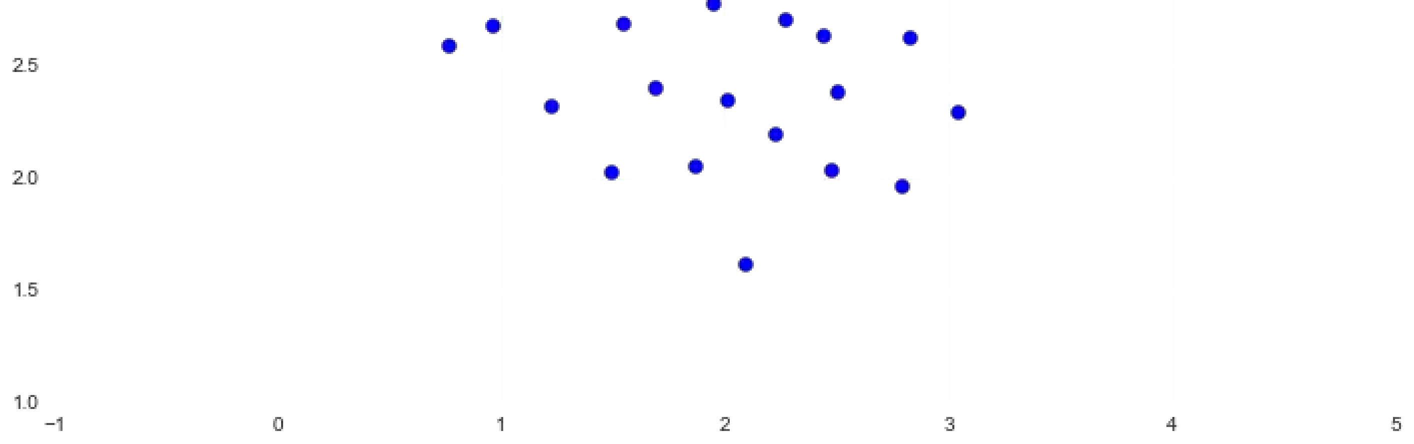
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
from scipy.io import loadmat
%matplotlib inline

raw_data = loadmat('data/ex6data1.mat')
raw_data
```

```
{'X': array([[ 1.9643  ,  4.5957  ],
        [ 2.2753  ,  3.8589  ],
        [ 2.9781  ,  4.5651  ],
        ...,
        [ 0.9044  ,  3.0198  ],
        [ 0.76615 ,  2.5899  ],
        [ 0.086405,  4.1045  ]]),
  '__globals__': [],
 '__header__': 'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Sun Nov 13
 '__version__': '1.0',
 'y': array([[1],
        [1],
        [1],
        ...,
        [0],
        [0],
        [1]], dtype=uint8)}
```

We'll visualize it as a scatter plot where the class label is denoted by a symbol ('+' for positive, 'o' for negative).

```
data = pd.DataFrame(raw_data['X'], columns=['X1', 'X2'])
data['y'] = raw_data['y']

positive = data[data['y'].isin([1])]
negative = data[data['y'].isin([0])]

fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(positive['X1'], positive['X2'], s=50, marker='x', label='Positive')
ax.scatter(negative['X1'], negative['X2'], s=50, marker='o', label='Negative')
ax.legend()
```





Notice that there is one outlier positive example that sits apart from the others. The classes are still linearly separable but it's a very tight fit. We're going to train a linear support vector machine to learn the class boundary. In this exercise we're not tasked with implementing an SVM from scratch, so I'm going to use the one built into scikit-learn.

```
from sklearn import svm
svc = svm.LinearSVC(C=1, loss='hinge', max_iter=1000)
svc
```

```
LinearSVC(C=1, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='hinge', max_iter=1000, multi_class='ovr',
     penalty='l2', random_state=None, tol=0.0001, verbose=0)
```

For the first experiment we'll use C=1 and see how it performs.

```
svc.fit(data[['X1', 'X2']], data['y'])
svc.score(data[['X1', 'X2']], data['y'])
```

```
0.9803921568627450
```

It appears that it mis-classified the outlier. Let's see what happens with a larger value of C

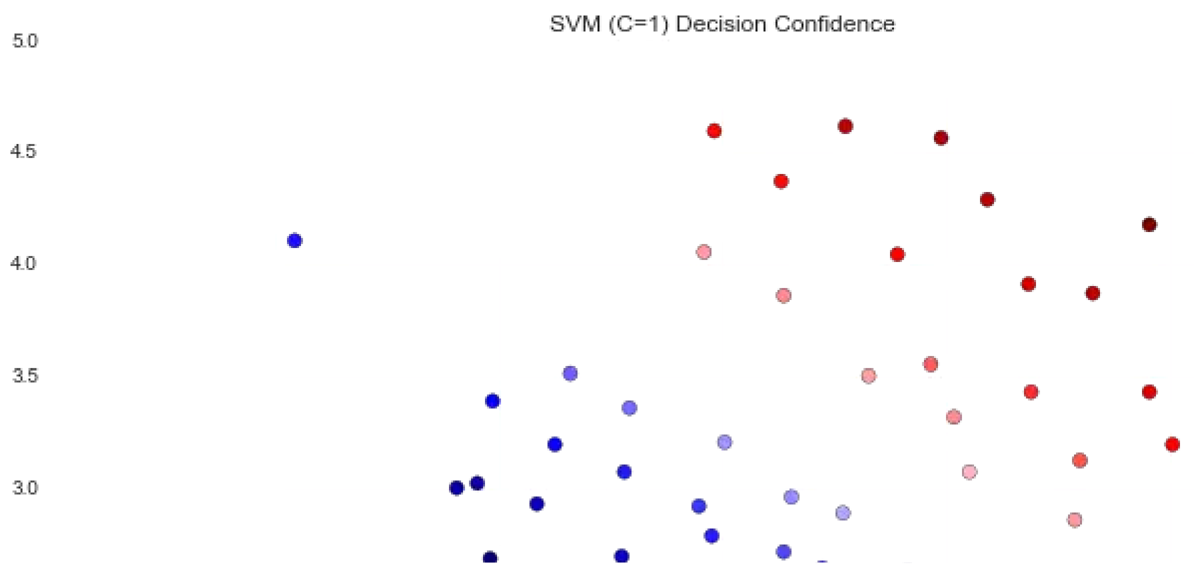
It appears that it mis-classified the outlier. Let's see what happens with a larger value of C.

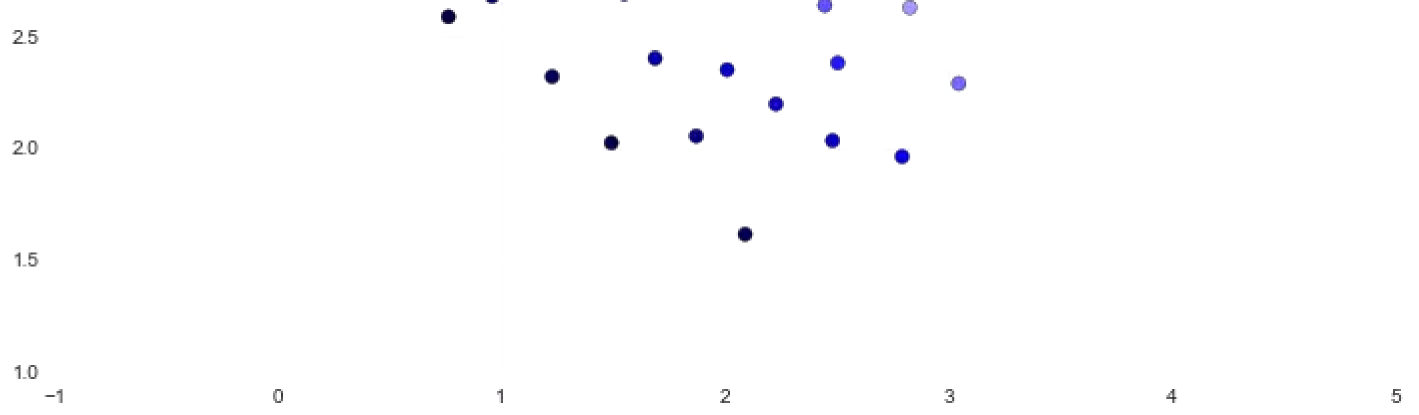
```
svc2 = svm.LinearSVC(C=100, loss='hinge', max_iter=1000)
svc2.fit(data[['X1', 'X2']], data['y'])
svc2.score(data[['X1', 'X2']], data['y'])
```

```
1.0
```

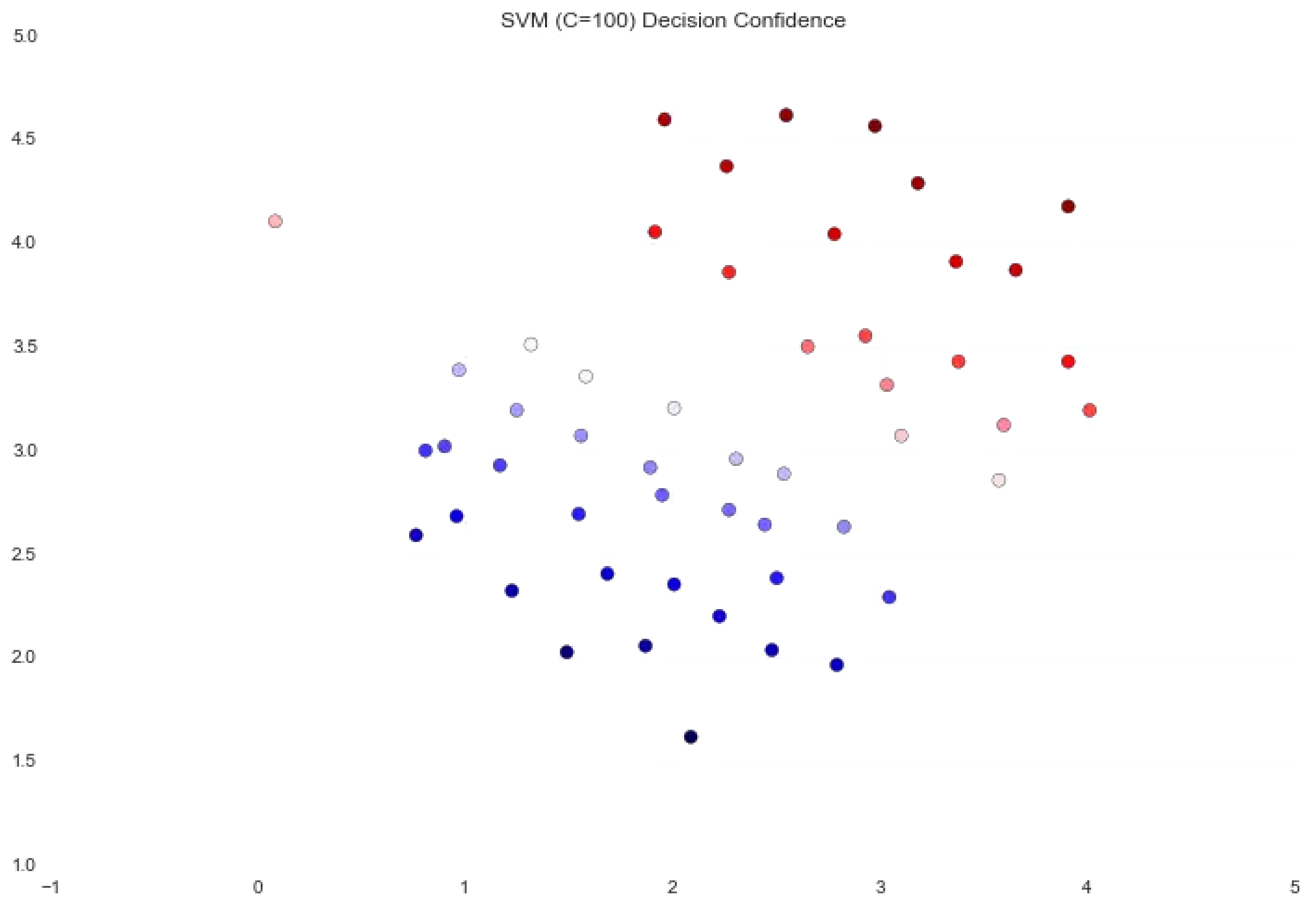
This time we got a perfect classification of the training data, however by increasing the value of C we've created a decision boundary that is no longer a natural fit for the data. We can visualize this by looking at the confidence level for each class prediction, which is a function of the point's distance from the hyperplane.

```
data['SVM 1 Confidence'] = svc.decision_function(data[['X1', 'X2']])

fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(data['X1'], data['X2'], s=50, c=data['SVM 1 Confidence'], cmap='seism
ax.set_title('SVM (C=1) Decision Confidence')
```





```
data['SVM 2 Confidence'] = svc2.decision_function(data[['X1', 'X2']])

fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(data['X1'], data['X2'], s=50, c=data['SVM 2 Confidence'], cmap='seism
ax.set_title('SVM (C=100) Decision Confidence')
```



The difference is a bit subtle but look at the color of the points near the boundary. In the firs image the points near the boundary are a strong red or blue, indicating that they're a solid distance from the hyperplane. This is not the case in the second image, where a number of points are nearly white, indicating that they are directly adjacent to the hyperplane.

Now we're going to move from a linear SVM to one that's capable of non-linear classification using kernels. We're first tasked with implementing a gaussian kernel function. Although scikit-learn has a gaussian kernel built in, for transparency we'll implement one from scratch.

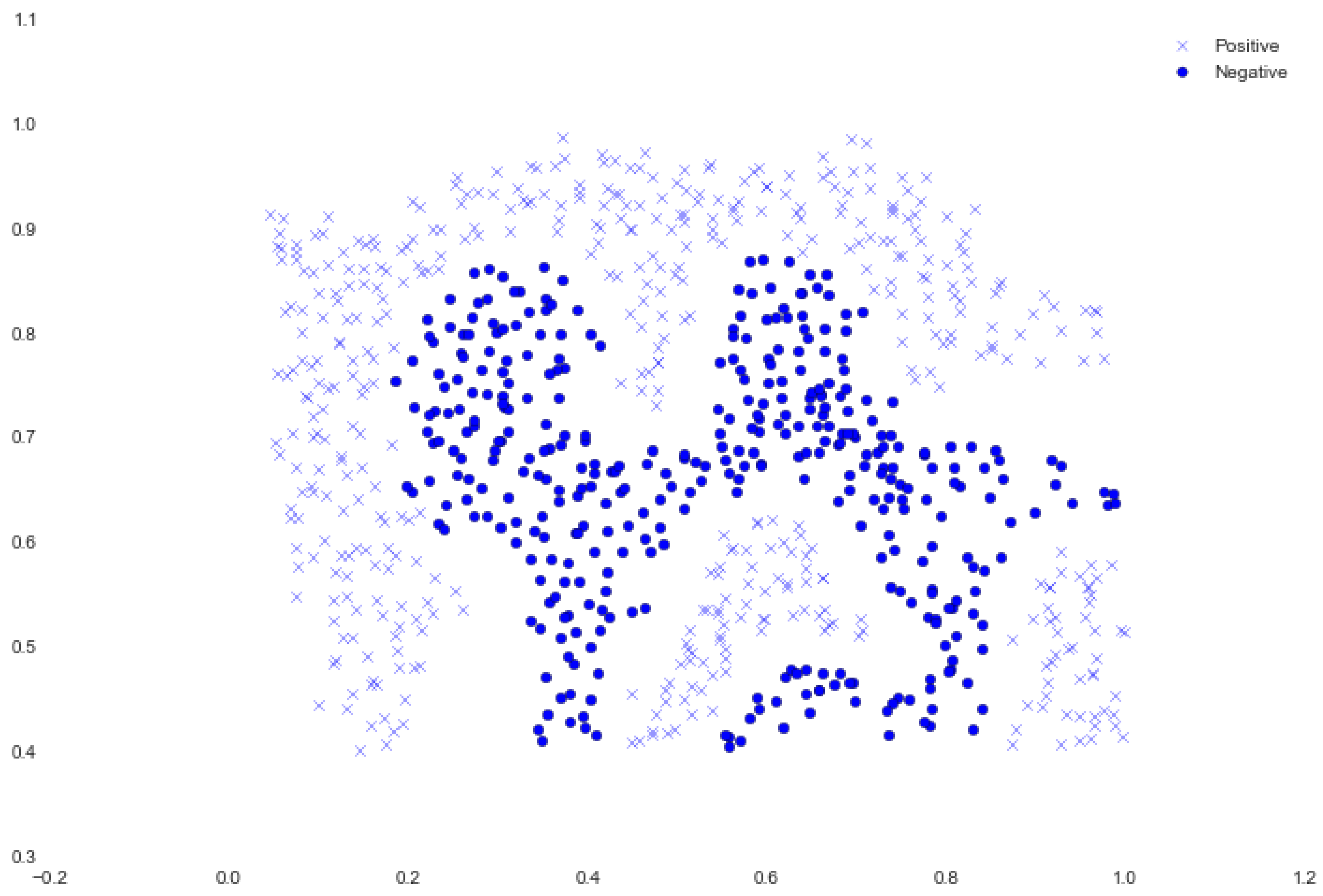
```
def gaussian_kernel(x1, x2, sigma):
    return np.exp(-(np.sum((x1 - x2) ** 2) / (2 * (sigma ** 2))))

x1 = np.array([1.0, 2.0, 1.0])
x2 = np.array([0.0, 4.0, -1.0])
sigma = 2
gaussian_kernel(x1, x2, sigma)
```

```
0.32465246735834974
```

That result matches the expected value from the exercise. Next we're going to examine another data set, this time with a non-linear decision boundary.

```
raw_data = loadmat('data/ex6data2.mat')

data = pd.DataFrame(raw_data['X'], columns=['X1', 'X2'])
data['y'] = raw_data['y']

positive = data[data['y'].isin([1])]
negative = data[data['y'].isin([0])]
```

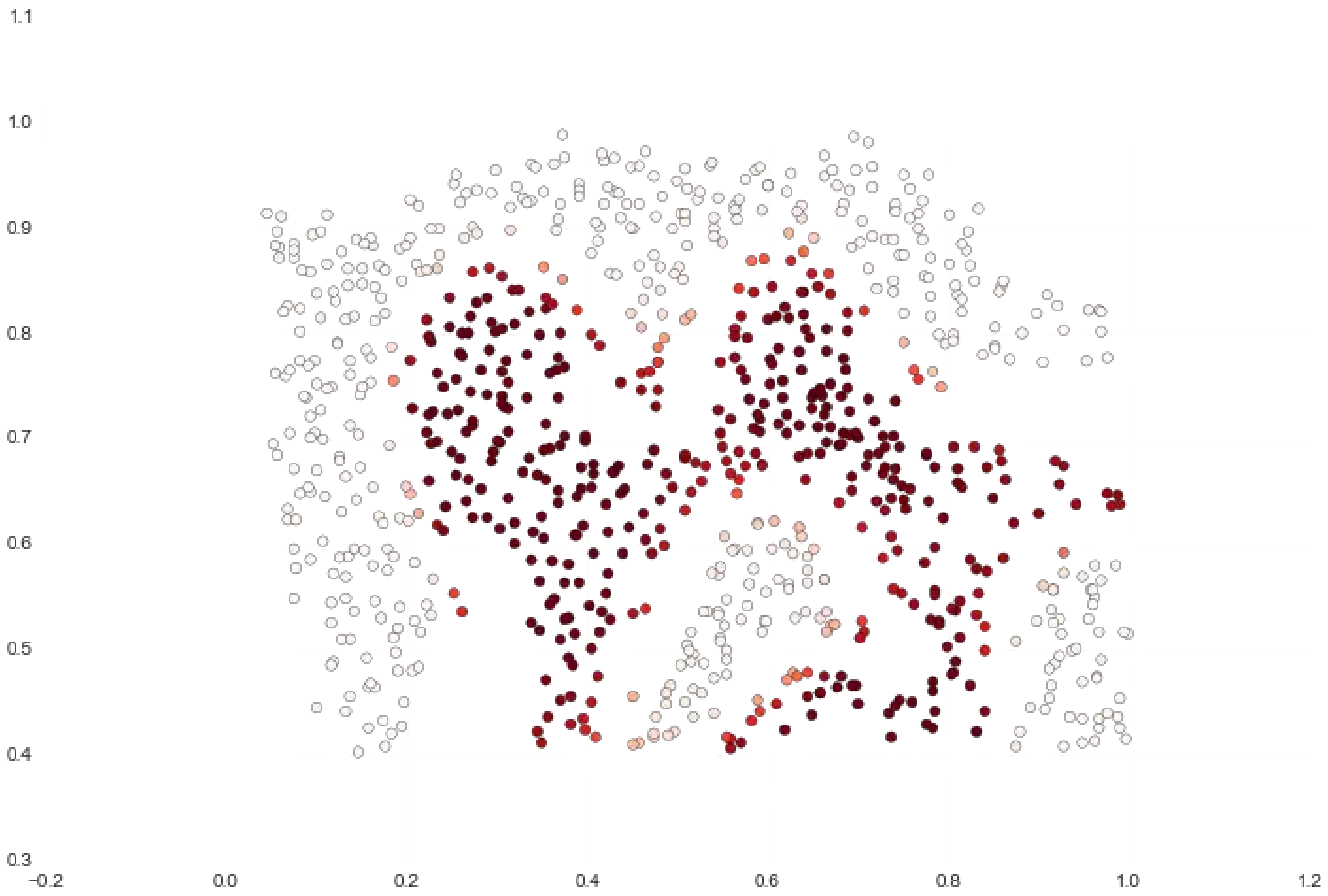
```python
fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(positive['X1'], positive['X2'], s=30, marker='x', label='Positive')
ax.scatter(negative['X1'], negative['X2'], s=30, marker='o', label='Negative')
ax.legend()
```



For this data set we'll build a support vector machine classifier using the built-in RBF kernel and examine its accuracy on the training data. To visualize the decision boundary, this time we'll shade the points based on the predicted probability that the instance has a negative class label. We'll see from the result that it gets most of them right.

```python
svc = svm.SVC(C=100, gamma=10, probability=True)
svc.fit(data[['X1', 'X2']], data['y'])
```

```
data['Probability'] = svc.predict_proba(data[['X1', 'X2']])[:,0]

fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(data['X1'], data['X2'], s=30, c=data['Probability'], cmap='Reds')
```



For the third data set we're given both training and validation sets and tasked with finding optimal hyper-parameters for an SVM model based on validation set performance. Although we could use scikit-learn's built-in grid search to do this quite easily, in the spirit of following the exercise directions we'll implement a simple grid search from scratch.

```
raw_data = loadmat('data/ex6data3.mat')

X = raw_data['X']
```

```
Xval = raw_data['Xval']
y = raw_data['y'].ravel()
yval = raw_data['yval'].ravel()

C_values = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100]
gamma_values = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100]

best_score = 0
best_params = {'C': None, 'gamma': None}

for C in C_values:
    for gamma in gamma_values:
        svc = svm.SVC(C=C, gamma=gamma)
        svc.fit(X, y)
        score = svc.score(Xval, yval)

        if score > best_score:
            best_score = score
            best_params['C'] = C
            best_params['gamma'] = gamma

best_score, best_params
```

```
(0.9649999999999997, {'C': 0.3, 'gamma': 100})
```

Now we'll move on to the last part of the exercise. In this part our objective is to use SVMs to build a spam filter. In the exercise text, there's a task involving some text pre-processing to get our data in a format suitable for an SVM to handle. However, the task is pretty trivial (mapping words to an ID from a dictionary that's provided for the exercise) and the rest of the pre-processing steps such as HTML removal, stemming, normalization etc. are already done.

Rather than reproduce these pre-processing steps, I'm going to skip ahead to the machine learning task which involves building a classifier from pre-processed train and test data sets consisting of spam and non-spam emails transformed to word occurance vectors.

```
spam_train = loadmat('data/spamTrain.mat')
spam_test = loadmat('data/spamTest.mat')

spam_train
```

```
{'X': array([[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 1, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]], dtype=uint8),
 '__globals__': [],
 '__header__': 'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Sun Nov 13 1
 '__version__': '1.0',
 'y': array([[1],
        [1],
        [0],
        ...,
        [1],
        [0],
        [0]], dtype=uint8)}
```

```
X = spam_train['X']
Xtest = spam_test['Xtest']
y = spam_train['y'].ravel()
ytest = spam_test['ytest'].ravel()

X.shape, y.shape, Xtest.shape, ytest.shape
```

```
((4000L, 1899L), (4000L,), (1000L, 1899L), (1000L,))
```

Each document has been converted to a vector with 1,899 dimensions corresponding to the 1,899 words in the vocabulary. The values are binary, indicating the presence or absence of the word in the document. At this point, training and evaluation are just a matter of fitting the testing the classifer.

```
svc = svm.SVC()
svc.fit(X, y)
print('Test accuracy = {0}%'.format(np.round(svc.score(Xtest, ytest) * 100, 2)))
```

```
Test accuracy = 95.3%
```

This result is with the default parameters. We could probably get it a bit higher using some parameter tuning, but 95% accuracy still isn't bad.

That concludes exercise 6! In the next exercise we'll perform clustering and image compression with K-Means and principal component analysis.

Follow me on twitter to get new post updates.

Follow @jdwittenauer

MACHINE LEARNING DATA SCIENCE DATA VISUALIZATION

AUTHOR

John Wittenauer