1 Support Vector Machines

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 do some pre-processing work on a set of raw emails and build a classifier on the processed emails using a SVM to determine if they are spam or not.

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.

```python
In [1]: 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
```

Out[1]: `{ 'X': array([[ 1.9643 , 4.5957 ],
       [ 2.2753 , 3.8589 ],
       [ 2.9781 , 3.8589 ],
       [ 2.932 , 3.5519 ],
       [ 3.5772 , 2.856 ],
       [ 4.015 , 3.1937 ],
       [ 3.3814 , 3.4291 ],
       [ 3.9113 , 4.1761 ],
       [ 2.7822 , 4.0431 ],
       [ 2.5518 , 4.6162 ],
       [ 3.3698 , 3.9101 ],
       [ 3.1048 , 3.0709 ],
       [ 1.9182 , 4.0534 ],
       [ 2.2638 , 4.3706 ],
       [ 2.6555 , 3.5008 ],
       [ 3.1855 , 4.2888 ],
       [ 3.6579 , 3.8692 ],
       [ 3.9113 , 3.4291 ],
       [ 3.6002 , 3.1221 ],
       [ 3.0357 , 3.3165 ],
       [ 1.5841 , 3.3575 ],
       [ 2.0103 , 3.2039 ],
       [ 1.9527 , 2.7843 ]],
      'y': array([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1])}`
[ 2.7523 , 2.7127 ],
[ 2.3099 , 2.9684 ],
[ 2.8283 , 2.6309 ],
[ 3.0473 , 2.2931 ],
[ 2.4827 , 2.0373 ],
[ 2.5057 , 2.3853 ],
[ 1.8721 , 2.0577 ],
[ 2.0103 , 2.3546 ],
[ 1.2269 , 2.3239 ],
[ 1.8951 , 2.9174 ],
[ 1.561 , 3.0709 ],
[ 1.5495 , 2.6923 ],
[ 1.6678 , 2.4057 ],
[ 1.4919 , 2.0271 ],
[ 0.962 , 2.682 ],
[ 1.1693 , 2.9276 ],
[ 0.8122 , 2.9992 ],
[ 0.9735 , 3.3881 ],
[ 1.25 , 3.1937 ],
[ 1.3191 , 3.5109 ],
[ 2.2292 , 2.201 ],
[ 2.4482 , 2.6411 ],
[ 2.7938 , 1.9656 ],
[ 2.091 , 1.6177 ],
[ 2.5403 , 2.8867 ],
[ 0.9044 , 3.0198 ],
[ 0.76615 , 2.5899 ],
[ 0.086405 , 4.1045 ]
['globals': []],
'__header__': b'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Sun Nov 13 14:28:43 2011',
'__version__': '1.0',
'y': array([[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1],
[1]],
[1],
[1],
[1],
[0],
[0],
[0],
[0],
[0]
We’ll visualize it as a scatter plot where the class label is denoted by a symbol (+ for positive, o for negative).

In [2]: 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()

Out[2]: <matplotlib.legend.Legend at 0x7fe873866e48>
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.

In [3]: from sklearn import svm
   svc = svm.LinearSVC(C=1, loss='hinge', max_iter=1000)

In [4]: svc.fit(data[‘X1’, ‘X2’], data[‘y’])
   svc.score(data[‘X1’, ‘X2’], data[‘y’])

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

In [5]: svc2 = svm.LinearSVC(C=100, loss=’hinge’, max_iter=1000)
   svc2.fit(data[‘X1’, ‘X2’], data[‘y’])
   svc2.score(data[‘X1’, ‘X2’], data[‘y’])

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.
In [6]: 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='seismic')
    ax.set_title('SVM (C=1) Decision Confidence')

Out[6]: <matplotlib.text.Text at 0x7fe870d97f28>

In [7]: 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='seismic')
    ax.set_title('SVM (C=100) Decision Confidence')

Out[7]: <matplotlib.text.Text at 0x7fe870d0e080>
The difference is a bit subtle but look at the color of the points near the boundary. If you’re following along in the exercise text, there’s a drawing where the decision boundary is shown as a line on the plot which helps make the difference a bit clearer.

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.

```python
In [8]: def gaussian_kernel(x1, x2, sigma):
   ...:     return np.exp(-(np.sum((x1 - x2) ** 2) / (2 * (sigma ** 2))))
   ...:
In [9]: 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)
   ...:
Out[9]: 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.

```python
In [10]: 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])]
```
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.

```
In [11]: svc = svm.SVC(C=100, gamma=10, probability=True)
svc
Out[11]: SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape=None, degree=3, gamma=10, kernel='rbf',
    max_iter=-1, probability=True, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

```
In [12]: svc.fit(data[['X1', 'X2']], data['y'])
svc.score(data[['X1', 'X2']], data['y'])
Out[12]: 0.9698725376593279
```

```
In [13]: data['Probability'] = svc.predict_proba(data[['X1', 'X2']])[:,0]
```

```python
    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.

```
In [14]: 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
```

8
best_params['C'] = C
best_params['gamma'] = gamma

best_score, best_params

Out[14]: (0.96499999999999997, {'C': 0.3, 'gamma': 100})

Now we'll move on to the second 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 occurrence vectors.

In [15]: spam_train = loadmat('data/spamTrain.mat')
spam_test = loadmat('data/spamTest.mat')

spam_train

Out[15]: {'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_': b'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Sun Nov 13 14:27:25 2011',
         '_version_': '1.0',
         'y': array([[1],
                     [1],
                     [0],
                     ...,
                     [1],
                     [0],
                     [0]], dtype=uint8)}

In [16]: 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

Out[16]: ((4000, 1899), (4000,), (1000, 1899), (1000,))

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 classifier.

In [17]: svc = svm.SVC()
svc.fit(X, y)
print("Training accuracy = {}\%").format(np.round(svc.score(X, y) * 100, 2))

Training accuracy = 94.4%
In [18]: 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.