# ml-ex6

August 4, 2017

## 0.1 Support vector machines

This exercise is described in ex6.pdf.

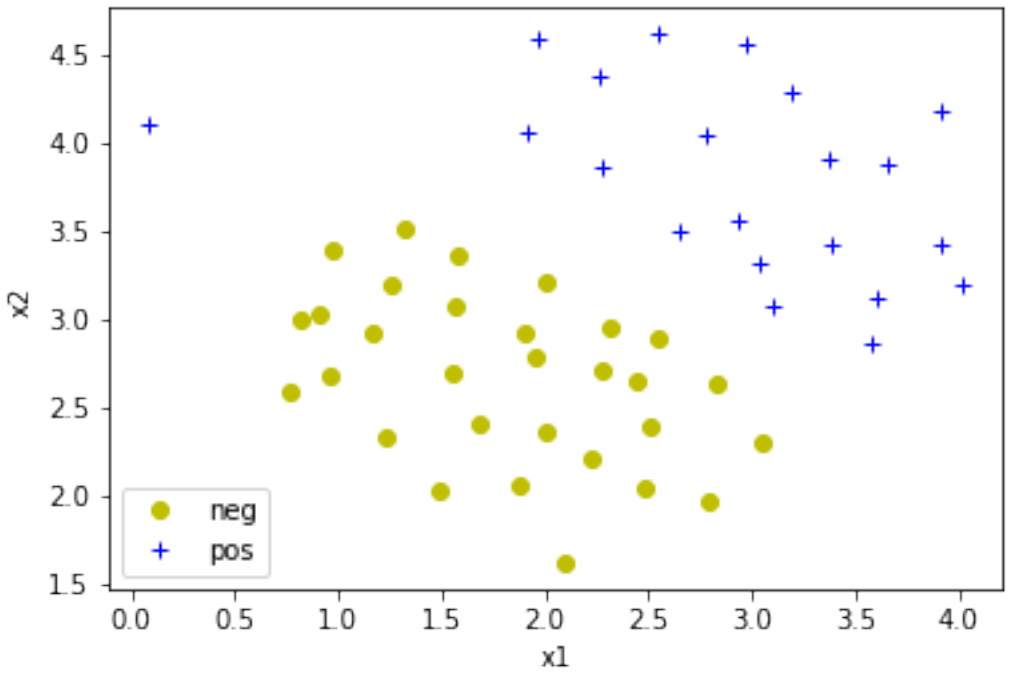
```
In [1]: import csv
        import numpy as np
        import pandas as pd
        import scipy.io as sio
        import matplotlib.pyplot as plt

        from oct2py import octave
        from sklearn.model_selection import GridSearchCV, PredefinedSplit
        from sklearn.svm import SVC, LinearSVC

        %matplotlib inline
```

### 0.1.1 SVM with linear kernel

```
In [2]: # Load dataset 1
        data = sio.loadmat('data/ml-ex6/ex6data1.mat')

In [3]: # Samples with features x1 and x2
        X = data['X']
        # Samples target class (0=neg, 1=pos)
        y = data['y'].ravel()

In [4]: def plot_data(X, y):
            '''Plots samples X and their class y in a 2D scatter plot.

            X must be an array of shape (m,2)
            y must be an array of shape (m,)
            '''

            plt.plot(X[y == 0,0], X[y == 0,1], 'yo', label='neg')
            plt.plot(X[y == 1,0], X[y == 1,1], 'b+', label='pos')
            plt.xlabel('x1')
            plt.ylabel('x2')
            plt.legend(loc='lower left')
```

```
    def plot_boundary(X, clf, level=0.0):
        '''Plots a decision boundary using trained classifier clf.

        The decision boundary is drawn at given level (default=0.0)
        in the range of X. X must be an array of shape (m,2).
        '''

        x1_min, x1_max = X[:, 0].min(), X[:, 0].max()
        x2_min, x2_max = X[:, 1].min(), X[:, 1].max()

        h = 0.01 # grid step size
        grid_x1, grid_x2 = np.meshgrid(np.arange(x1_min, x1_max, h),
                                       np.arange(x2_min, x2_max, h))

        grid_y = clf.predict(np.c_[grid_x1.ravel(), grid_x2.ravel()])
        grid_y = grid_y.reshape(grid_x1.shape)

        x1_extra = (x1_max - x1_min) / 50
        x2_extra = (x2_max - x2_min) / 50

        plt.xlim(x1_min - x1_extra, x1_max + x1_extra)
        plt.ylim(x2_min - x2_extra, x2_max + x2_extra)

        plt.contour(grid_x1, grid_x2, grid_y, levels=[level])

    def plot_support_vectors(clf):
        '''Plot the support vectors from trained support vector classifier clf.

        Only the support vectors of the first two classes are drawn.
        '''

        # Obtain support vectors for first two classes
        sv_neg, sv_pos = np.vsplit(clf.support_vectors_, np.cumsum(clf.n_support_)[0:1])

        # Plot support vectors
        plt.plot(sv_neg[:,0], sv_neg[:,1], 'r+', label='neg SVs', alpha=0.5)
        plt.plot(sv_pos[:,0], sv_pos[:,1], 'rx', label='pos SVs', alpha=0.5)
        plt.legend()
```

In [5]:
```
# Plot dataset 1
plot_data(X, y)
```



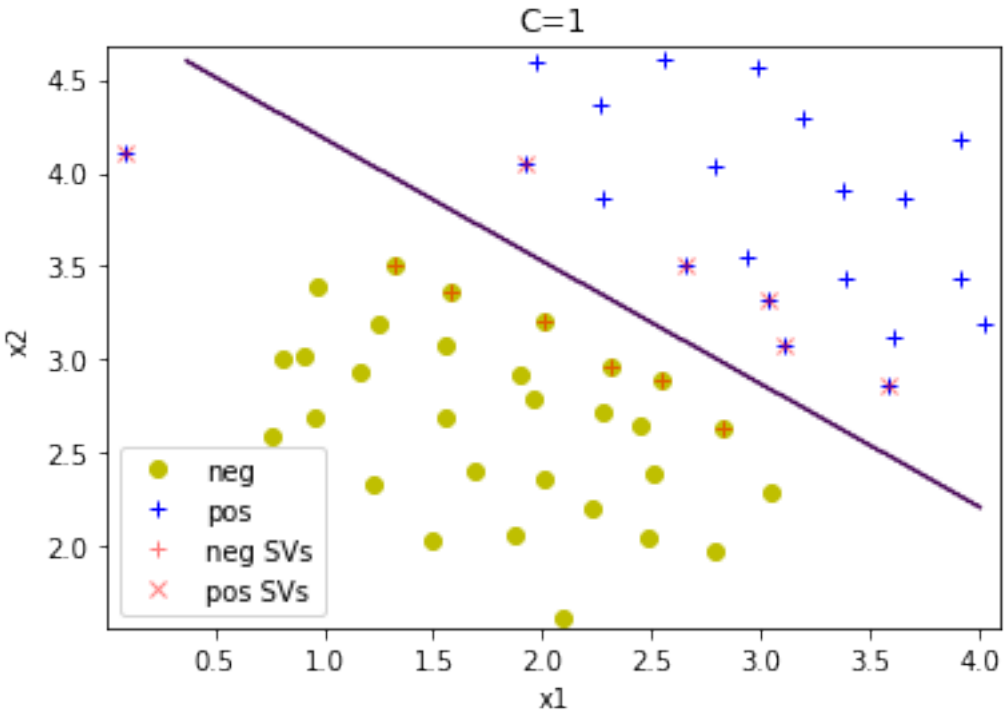

```
In [6]: # Train a support vector classifier with C=1 using a linear kernel.
        # SVC uses libsvm. For scaling to a large number of samples consider
        # using LinearSVC which uses liblinear instead (see docs for details).
        clf = SVC(C=1, kernel='linear')
        clf.fit(X, y)
```

```
Out[6]: 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)
```

C=1 misclassifies the outlier as shown in the following figure:

```
In [7]: plot_data(X, y)
        plot_boundary(X, clf)
        plot_support_vectors(clf)
        plt.title('C=1')
```

```
Out[7]: <matplotlib.text.Text at 0x107ffa470>
```



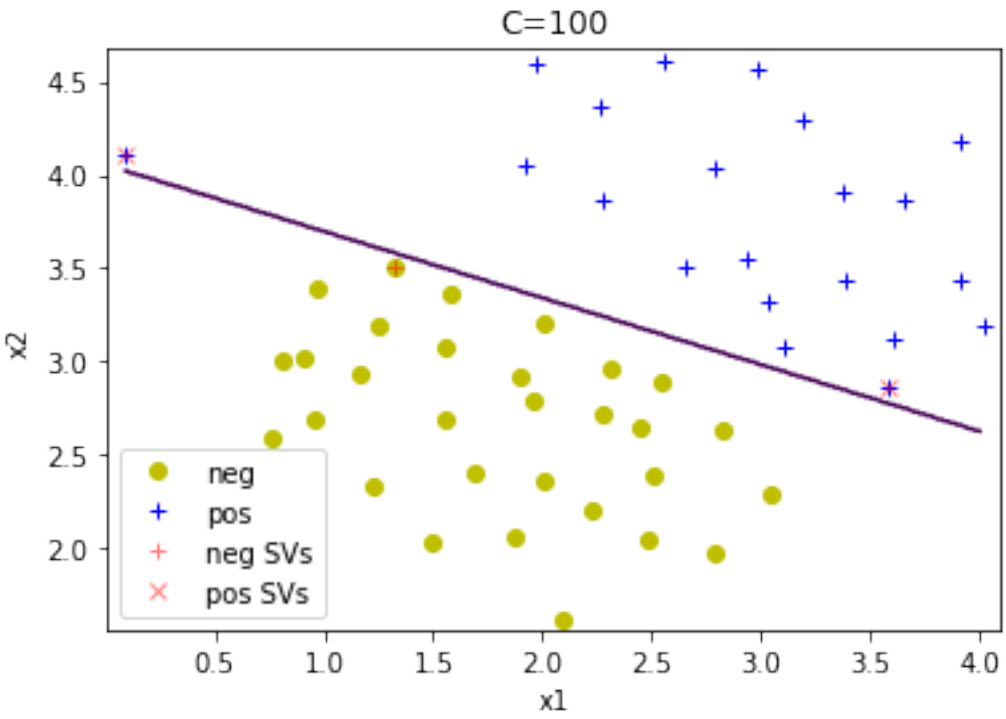

```
In [8]: # Train a support vector classifier with C=100 using a linear kernel.
        clf = SVC(C=100, kernel='linear')
        clf.fit(X, y)
```

```
Out[8]: SVC(C=100, 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)
```

C=100 correctly classifies the outlier as shown in the following figure (less regularization = higher variance, lower bias):

```
In [9]: plot_data(X, y)
        plot_boundary(X, clf)
        plot_support_vectors(clf)
        plt.title('C=100')
```

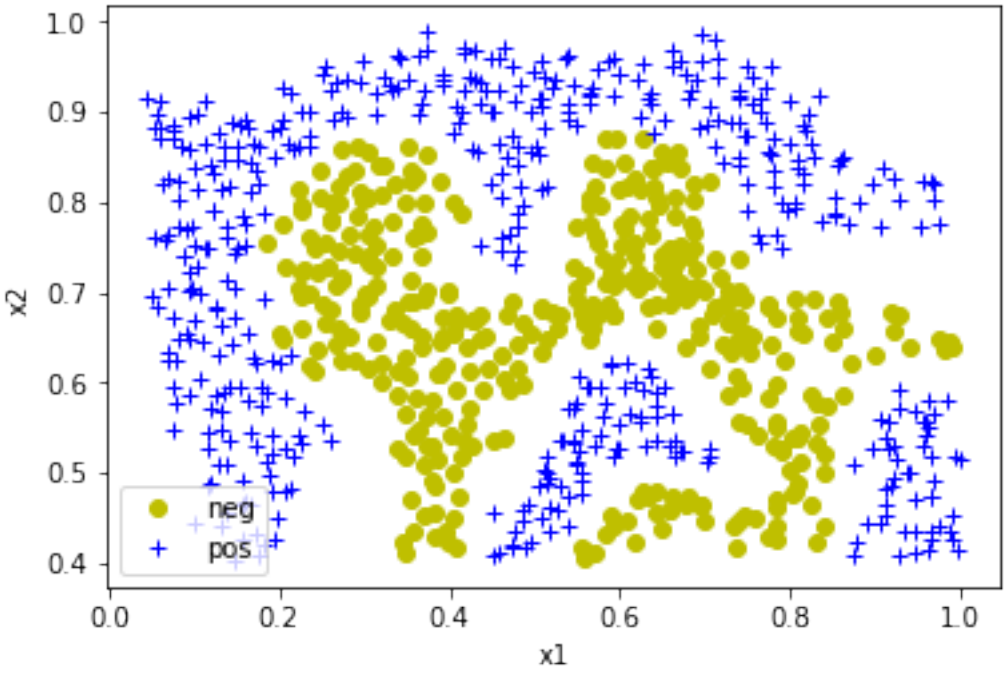
```
Out[9]: <matplotlib.text.Text at 0x108719a58>
```




### 0.1.2 SVM with Gaussian kernel

An RBF kernel with gamma=1/sigma2 is know as Gaussian kernel of variance sigma2.

```
In [10]: # Load dataset 2
         data = sio.loadmat('data/ml-ex6/ex6data2.mat')

In [11]: # Samples with features x1 and x2
         X = data['X']
         # Samples target class (0=neg, 1=pos)
         y = data['y'].ravel()

In [12]: # Plot dataset 2
         plot_data(X, y)
```

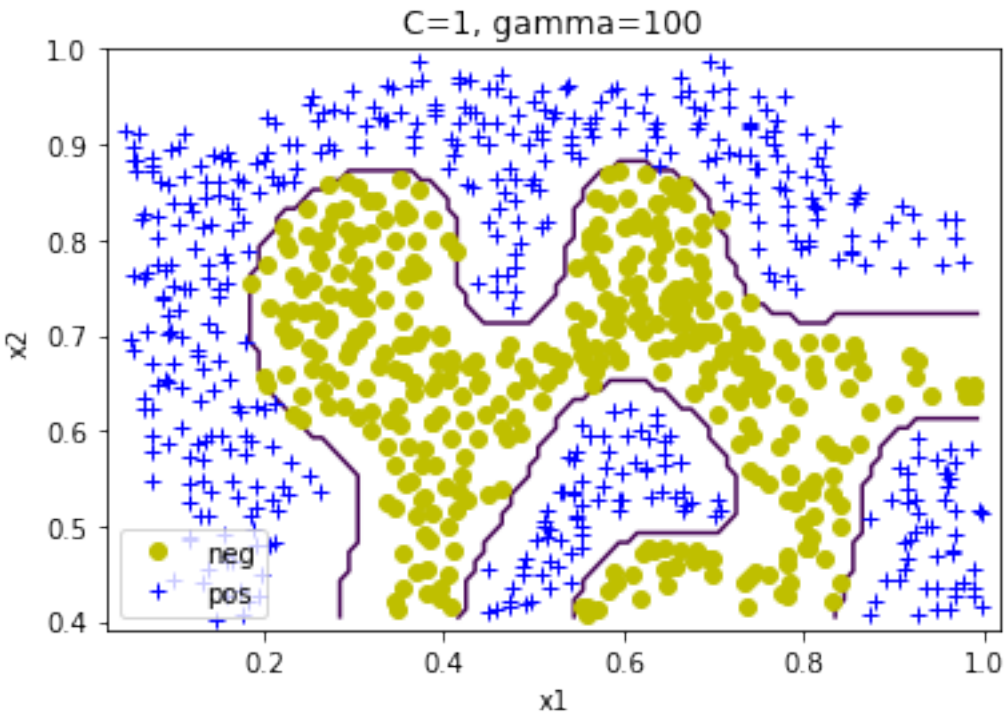



In [13]:
```
# Train a support vector classifier with C=1 using an RBF kernel.
# In ex6.m (original Octave exercise code), sigma=0.1, hence we
# set gamma to 100 (=1/sigma**2)
clf = SVC(C=1, kernel='rbf', gamma=100)
clf.fit(X, y)
```

Out[13]:
```
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma=100, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

In [14]:
```
plot_data(X, y)
plot_boundary(X, clf)
plt.title('C=1, gamma=100')
```

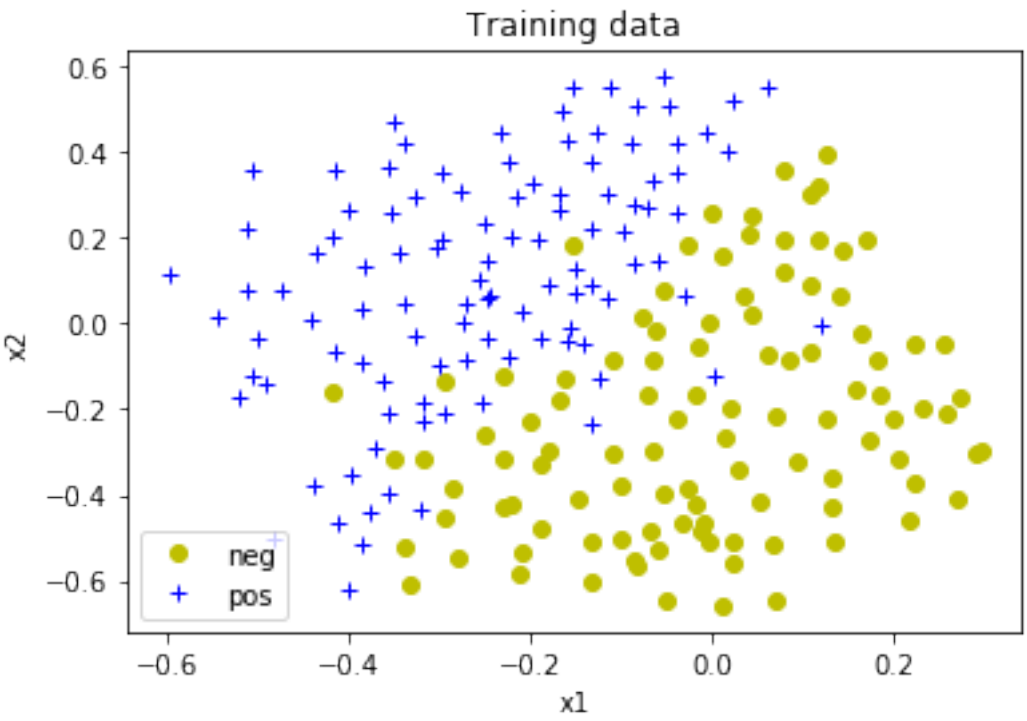
Out[14]:
```
<matplotlib.text.Text at 0x108de3710>
```




In [15]:
```
# Load dataset 3
data = sio.loadmat('data/ml-ex6/ex6data3.mat')

# Training data
X_train_0 = data['X']
y_train_0 = data['y'].ravel()

# Validation data
X_cv_0 = data['Xval']
y_cv_0 = data['yval'].ravel()
```

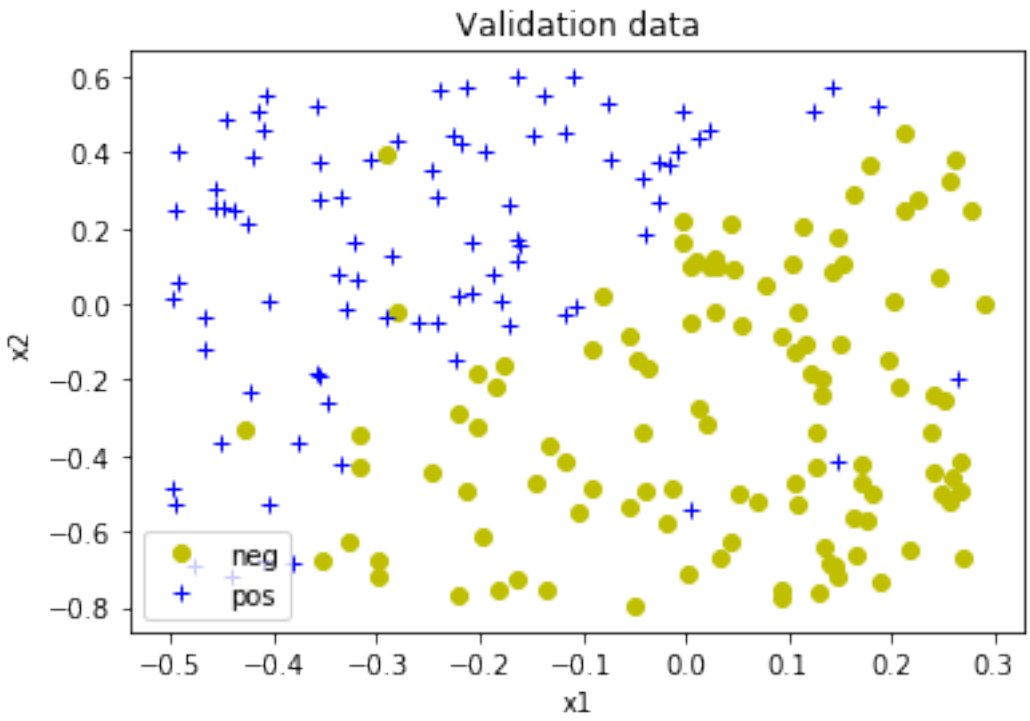
In [16]:
```
plot_data(X_train_0, y_train_0)
plt.title('Training data')
```

Out[16]:
```
<matplotlib.text.Text at 0x108f56f60>
```




```
In [17]: plot_data(X_cv_0, y_cv_0)
         plt.title('Validation data')
```

```
Out[17]: <matplotlib.text.Text at 0x108ffc0b8>
```




In [18]:
```
# Concatenate training and validation data
X = np.concatenate([X_train_0, X_cv_0])
y = np.concatenate([y_train_0, y_cv_0])

num_train = X_train_0.shape[0]
num_cv = X_cv_0.shape[0]

# Create a cross validator that selects the pre-defined
# validation dataset from the concatenated dataset.
cv_fold = np.empty(num_train + num_cv, dtype='int8')
cv_fold[:num_train] = -1
cv_fold[num_train:] = 0
cv = PredefinedSplit(cv_fold)
```

In [19]:
```
# Values for grid search (see description in ex6.pdf)
grid = np.array([0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30])

# Grid values for C
grid_C = grid
# Grid values for gamma
grid_gamma = 1 / (grid ** 2)
```

In [20]:
```
# Classifier used for grid search
clf = SVC(kernel='rbf')
```

```
# Grid search to find the best C and gamma values using the predefined
# training and validation set.
gs = GridSearchCV(clf, param_grid={'C':grid_C, 'gamma':grid_gamma}, cv=cv)
gs.fit(X, y)
```

Out[20]:
```
GridSearchCV(cv=PredefinedSplit(test_fold=array([-1, -1, ...,  0,  0])),
       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='rbf',
  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': array([  1.00000e-02,   3.00000e-02,   1.00000e-01,   3.00000e-
         1.00000e+00,   3.00000e+00,   1.00000e+01,   3.00000e+01]), 'gamma': array([
         1.00000e+00,   1.11111e-01,   1.00000e-02,   1.11111e-03])},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=None, verbose=0)
```

In [21]:
```
# Display grid search results in a pandas DataFrame
pd.DataFrame(gs.cv_results_)
```

Out[21]:

| | mean_fit_time | mean_score_time | mean_test_score | mean_train_score | param_C | \ |
|---|---|---|---|---|---|---|
| 0 | 0.003872 | 0.001880 | 0.435 | 0.502370 | 0.01 | |
| 1 | 0.003072 | 0.001746 | 0.435 | 0.502370 | 0.01 | |
| 2 | 0.003044 | 0.001766 | 0.435 | 0.502370 | 0.01 | |
| 3 | 0.003155 | 0.001521 | 0.435 | 0.502370 | 0.01 | |
| 4 | 0.001952 | 0.001278 | 0.435 | 0.502370 | 0.01 | |
| 5 | 0.002496 | 0.001414 | 0.435 | 0.502370 | 0.01 | |
| 6 | 0.001969 | 0.001275 | 0.435 | 0.502370 | 0.01 | |
| 7 | 0.001987 | 0.001479 | 0.435 | 0.502370 | 0.01 | |
| 8 | 0.002463 | 0.001579 | 0.435 | 0.502370 | 0.03 | |
| 9 | 0.002587 | 0.001685 | 0.435 | 0.502370 | 0.03 | |
| 10 | 0.002254 | 0.001318 | 0.435 | 0.502370 | 0.03 | |
| 11 | 0.002467 | 0.001426 | 0.900 | 0.876777 | 0.03 | |
| 12 | 0.002076 | 0.001698 | 0.795 | 0.819905 | 0.03 | |
| 13 | 0.002307 | 0.001430 | 0.435 | 0.502370 | 0.03 | |
| 14 | 0.002448 | 0.001467 | 0.435 | 0.502370 | 0.03 | |
| 15 | 0.002211 | 0.001579 | 0.435 | 0.502370 | 0.03 | |
| 16 | 0.002431 | 0.001407 | 0.435 | 0.502370 | 0.1 | |
| 17 | 0.002848 | 0.001708 | 0.435 | 0.502370 | 0.1 | |
| 18 | 0.002570 | 0.001626 | 0.860 | 0.919431 | 0.1 | |
| 19 | 0.001493 | 0.000929 | 0.930 | 0.900474 | 0.1 | |
| 20 | 0.001622 | 0.001073 | 0.845 | 0.862559 | 0.1 | |
| 21 | 0.001955 | 0.001278 | 0.535 | 0.511848 | 0.1 | |
| 22 | 0.001963 | 0.001276 | 0.435 | 0.502370 | 0.1 | |
| 23 | 0.002036 | 0.001344 | 0.435 | 0.502370 | 0.1 | |
| 24 | 0.002089 | 0.001361 | 0.435 | 0.502370 | 0.3 | |

```
25       0.002789         0.001848           0.465          0.668246     0.3
26       0.003104         0.001402           0.965          0.952607     0.3
27       0.001438         0.000941           0.950          0.933649     0.3
28       0.001585         0.001066           0.910          0.876777     0.3
29       0.002141         0.001394           0.825          0.819905     0.3
..            ...              ...             ...               ...     ...
34       0.002750         0.001006           0.965          0.943128       1
35       0.001175         0.000744           0.950          0.938389       1
36       0.001229         0.000823           0.940          0.919431       1
37       0.001528         0.001029           0.870          0.867299       1
38       0.001968         0.001257           0.500          0.502370       1
39       0.001896         0.001238           0.435          0.502370       1
40       0.002543         0.001232           0.555          1.000000       3
41       0.003045         0.001328           0.880          1.000000       3
42       0.002206         0.000842           0.945          0.957346       3
43       0.001106         0.000660           0.955          0.938389       3
44       0.001100         0.000694           0.925          0.924171       3
45       0.001211         0.000815           0.915          0.876777       3
46       0.001664         0.001094           0.820          0.824645       3
47       0.001753         0.001149           0.435          0.502370       3
48       0.002299         0.001073           0.555          1.000000      10
49       0.002759         0.001222           0.880          1.000000      10
50       0.001894         0.000710           0.945          0.962085      10
51       0.001242         0.000555           0.960          0.943128      10
52       0.000968         0.000581           0.925          0.928910      10
53       0.000985         0.000659           0.930          0.924171      10
54       0.001307         0.000852           0.865          0.867299      10
55       0.001613         0.001072           0.590          0.530806      10
56       0.002117         0.001000           0.555          1.000000      30
57       0.002576         0.001141           0.880          1.000000      30
58       0.001867         0.000697           0.920          0.985782      30
59       0.001539         0.000694           0.960          0.943128      30
60       0.001156         0.000616           0.935          0.928910      30
61       0.001067         0.000627           0.935          0.924171      30
62       0.001079         0.000731           0.915          0.876777      30
63       0.001427         0.000939           0.820          0.819905      30

   param_gamma                                  params  rank_test_score  \
0        10000           {'C': 0.01, 'gamma': 10000.0}               42
1      1111.11     {'C': 0.01, 'gamma': 1111.11111111}               42
2          100             {'C': 0.01, 'gamma': 100.0}               42
3      11.1111     {'C': 0.01, 'gamma': 11.1111111111}               42
4            1               {'C': 0.01, 'gamma': 1.0}               42
5     0.111111    {'C': 0.01, 'gamma': 0.111111111111}               42
6         0.01              {'C': 0.01, 'gamma': 0.01}               42
7   0.00111111  {'C': 0.01, 'gamma': 0.00111111111111}               42
8        10000           {'C': 0.03, 'gamma': 10000.0}               42
9      1111.11     {'C': 0.03, 'gamma': 1111.11111111}               42
```

| | | | |
|---|---|---|---|
| 10 | 100 | {'C': 0.03, 'gamma': 100.0} | 42 |
| 11 | 11.1111 | {'C': 0.03, 'gamma': 11.1111111111} | 21 |
| 12 | 1 | {'C': 0.03, 'gamma': 1.0} | 33 |
| 13 | 0.111111 | {'C': 0.03, 'gamma': 0.111111111111} | 42 |
| 14 | 0.01 | {'C': 0.03, 'gamma': 0.01} | 42 |
| 15 | 0.00111111 | {'C': 0.03, 'gamma': 0.00111111111111} | 42 |
| 16 | 10000 | {'C': 0.1, 'gamma': 10000.0} | 42 |
| 17 | 1111.11 | {'C': 0.1, 'gamma': 1111.11111111} | 42 |
| 18 | 100 | {'C': 0.1, 'gamma': 100.0} | 28 |
| 19 | 11.1111 | {'C': 0.1, 'gamma': 11.1111111111} | 13 |
| 20 | 1 | {'C': 0.1, 'gamma': 1.0} | 29 |
| 21 | 0.111111 | {'C': 0.1, 'gamma': 0.111111111111} | 39 |
| 22 | 0.01 | {'C': 0.1, 'gamma': 0.01} | 42 |
| 23 | 0.00111111 | {'C': 0.1, 'gamma': 0.00111111111111} | 42 |
| 24 | 10000 | {'C': 0.3, 'gamma': 10000.0} | 42 |
| 25 | 1111.11 | {'C': 0.3, 'gamma': 1111.11111111} | 41 |
| 26 | 100 | {'C': 0.3, 'gamma': 100.0} | 1 |
| 27 | 11.1111 | {'C': 0.3, 'gamma': 11.1111111111} | 6 |
| 28 | 1 | {'C': 0.3, 'gamma': 1.0} | 20 |
| 29 | 0.111111 | {'C': 0.3, 'gamma': 0.111111111111} | 30 |
| .. | ... | ... | ... |
| 34 | 100 | {'C': 1.0, 'gamma': 100.0} | 1 |
| 35 | 11.1111 | {'C': 1.0, 'gamma': 11.1111111111} | 6 |
| 36 | 1 | {'C': 1.0, 'gamma': 1.0} | 10 |
| 37 | 0.111111 | {'C': 1.0, 'gamma': 0.111111111111} | 26 |
| 38 | 0.01 | {'C': 1.0, 'gamma': 0.01} | 40 |
| 39 | 0.00111111 | {'C': 1.0, 'gamma': 0.00111111111111} | 42 |
| 40 | 10000 | {'C': 3.0, 'gamma': 10000.0} | 35 |
| 41 | 1111.11 | {'C': 3.0, 'gamma': 1111.11111111} | 23 |
| 42 | 100 | {'C': 3.0, 'gamma': 100.0} | 8 |
| 43 | 11.1111 | {'C': 3.0, 'gamma': 11.1111111111} | 5 |
| 44 | 1 | {'C': 3.0, 'gamma': 1.0} | 15 |
| 45 | 0.111111 | {'C': 3.0, 'gamma': 0.111111111111} | 18 |
| 46 | 0.01 | {'C': 3.0, 'gamma': 0.01} | 31 |
| 47 | 0.00111111 | {'C': 3.0, 'gamma': 0.00111111111111} | 42 |
| 48 | 10000 | {'C': 10.0, 'gamma': 10000.0} | 35 |
| 49 | 1111.11 | {'C': 10.0, 'gamma': 1111.11111111} | 23 |
| 50 | 100 | {'C': 10.0, 'gamma': 100.0} | 8 |
| 51 | 11.1111 | {'C': 10.0, 'gamma': 11.1111111111} | 3 |
| 52 | 1 | {'C': 10.0, 'gamma': 1.0} | 15 |
| 53 | 0.111111 | {'C': 10.0, 'gamma': 0.111111111111} | 13 |
| 54 | 0.01 | {'C': 10.0, 'gamma': 0.01} | 27 |
| 55 | 0.00111111 | {'C': 10.0, 'gamma': 0.00111111111111} | 34 |
| 56 | 10000 | {'C': 30.0, 'gamma': 10000.0} | 35 |
| 57 | 1111.11 | {'C': 30.0, 'gamma': 1111.11111111} | 23 |
| 58 | 100 | {'C': 30.0, 'gamma': 100.0} | 17 |
| 59 | 11.1111 | {'C': 30.0, 'gamma': 11.1111111111} | 3 |
| 60 | 1 | {'C': 30.0, 'gamma': 1.0} | 11 |

```
61    0.111111    {'C': 30.0, 'gamma': 0.111111111111}              11
62        0.01              {'C': 30.0, 'gamma': 0.01}              18
63  0.00111111  {'C': 30.0, 'gamma': 0.00111111111111}              31

    split0_test_score  split0_train_score  std_fit_time  std_score_time  \
0               0.435           0.502370           0.0             0.0
1               0.435           0.502370           0.0             0.0
2               0.435           0.502370           0.0             0.0
3               0.435           0.502370           0.0             0.0
4               0.435           0.502370           0.0             0.0
5               0.435           0.502370           0.0             0.0
6               0.435           0.502370           0.0             0.0
7               0.435           0.502370           0.0             0.0
8               0.435           0.502370           0.0             0.0
9               0.435           0.502370           0.0             0.0
10              0.435           0.502370           0.0             0.0
11              0.900           0.876777           0.0             0.0
12              0.795           0.819905           0.0             0.0
13              0.435           0.502370           0.0             0.0
14              0.435           0.502370           0.0             0.0
15              0.435           0.502370           0.0             0.0
16              0.435           0.502370           0.0             0.0
17              0.435           0.502370           0.0             0.0
18              0.860           0.919431           0.0             0.0
19              0.930           0.900474           0.0             0.0
20              0.845           0.862559           0.0             0.0
21              0.535           0.511848           0.0             0.0
22              0.435           0.502370           0.0             0.0
23              0.435           0.502370           0.0             0.0
24              0.435           0.502370           0.0             0.0
25              0.465           0.668246           0.0             0.0
26              0.965           0.952607           0.0             0.0
27              0.950           0.933649           0.0             0.0
28              0.910           0.876777           0.0             0.0
29              0.825           0.819905           0.0             0.0
..                ...                ...           ...             ...
34              0.965           0.943128           0.0             0.0
35              0.950           0.938389           0.0             0.0
36              0.940           0.919431           0.0             0.0
37              0.870           0.867299           0.0             0.0
38              0.500           0.502370           0.0             0.0
39              0.435           0.502370           0.0             0.0
40              0.555           1.000000           0.0             0.0
41              0.880           1.000000           0.0             0.0
42              0.945           0.957346           0.0             0.0
43              0.955           0.938389           0.0             0.0
44              0.925           0.924171           0.0             0.0
45              0.915           0.876777           0.0             0.0
```

| | | | | |
|---|---|---|---|---|
| 46 | 0.820 | 0.824645 | 0.0 | 0.0 |
| 47 | 0.435 | 0.502370 | 0.0 | 0.0 |
| 48 | 0.555 | 1.000000 | 0.0 | 0.0 |
| 49 | 0.880 | 1.000000 | 0.0 | 0.0 |
| 50 | 0.945 | 0.962085 | 0.0 | 0.0 |
| 51 | 0.960 | 0.943128 | 0.0 | 0.0 |
| 52 | 0.925 | 0.928910 | 0.0 | 0.0 |
| 53 | 0.930 | 0.924171 | 0.0 | 0.0 |
| 54 | 0.865 | 0.867299 | 0.0 | 0.0 |
| 55 | 0.590 | 0.530806 | 0.0 | 0.0 |
| 56 | 0.555 | 1.000000 | 0.0 | 0.0 |
| 57 | 0.880 | 1.000000 | 0.0 | 0.0 |
| 58 | 0.920 | 0.985782 | 0.0 | 0.0 |
| 59 | 0.960 | 0.943128 | 0.0 | 0.0 |
| 60 | 0.935 | 0.928910 | 0.0 | 0.0 |
| 61 | 0.935 | 0.924171 | 0.0 | 0.0 |
| 62 | 0.915 | 0.876777 | 0.0 | 0.0 |
| 63 | 0.820 | 0.819905 | 0.0 | 0.0 |

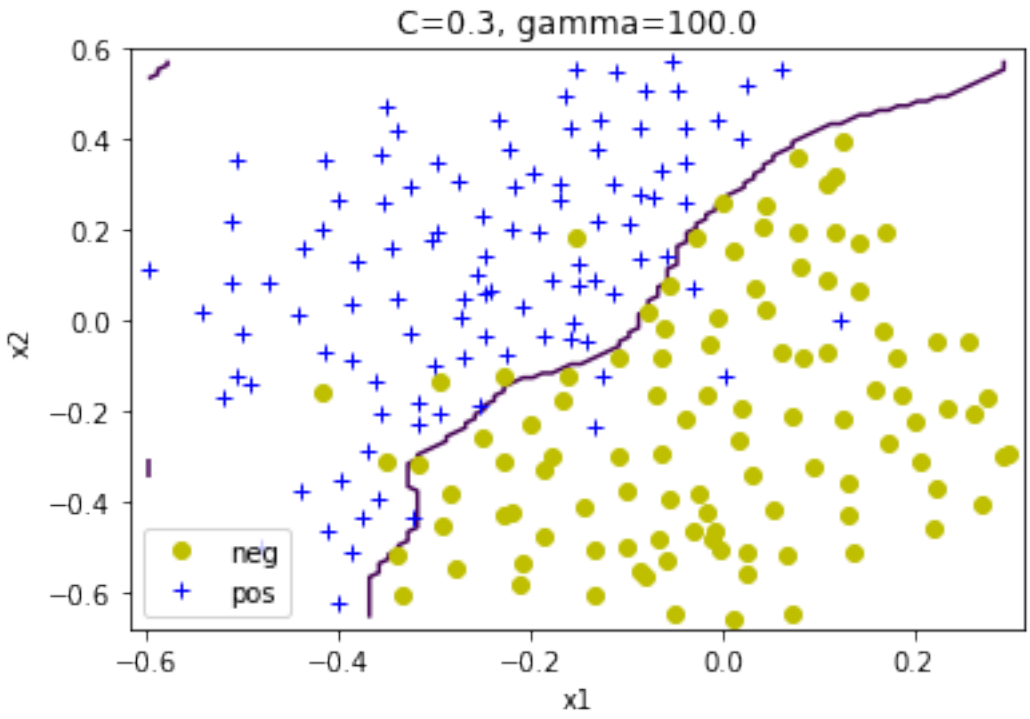
| | std_test_score | std_train_score |
|---|---|---|
| 0 | 0.0 | 0.0 |
| 1 | 0.0 | 0.0 |
| 2 | 0.0 | 0.0 |
| 3 | 0.0 | 0.0 |
| 4 | 0.0 | 0.0 |
| 5 | 0.0 | 0.0 |
| 6 | 0.0 | 0.0 |
| 7 | 0.0 | 0.0 |
| 8 | 0.0 | 0.0 |
| 9 | 0.0 | 0.0 |
| 10 | 0.0 | 0.0 |
| 11 | 0.0 | 0.0 |
| 12 | 0.0 | 0.0 |
| 13 | 0.0 | 0.0 |
| 14 | 0.0 | 0.0 |
| 15 | 0.0 | 0.0 |
| 16 | 0.0 | 0.0 |
| 17 | 0.0 | 0.0 |
| 18 | 0.0 | 0.0 |
| 19 | 0.0 | 0.0 |
| 20 | 0.0 | 0.0 |
| 21 | 0.0 | 0.0 |
| 22 | 0.0 | 0.0 |
| 23 | 0.0 | 0.0 |
| 24 | 0.0 | 0.0 |
| 25 | 0.0 | 0.0 |
| 26 | 0.0 | 0.0 |
| 27 | 0.0 | 0.0 |

```
28              0.0             0.0
29              0.0             0.0
..              ...             ...
34              0.0             0.0
35              0.0             0.0
36              0.0             0.0
37              0.0             0.0
38              0.0             0.0
39              0.0             0.0
40              0.0             0.0
41              0.0             0.0
42              0.0             0.0
43              0.0             0.0
44              0.0             0.0
45              0.0             0.0
46              0.0             0.0
47              0.0             0.0
48              0.0             0.0
49              0.0             0.0
50              0.0             0.0
51              0.0             0.0
52              0.0             0.0
53              0.0             0.0
54              0.0             0.0
55              0.0             0.0
56              0.0             0.0
57              0.0             0.0
58              0.0             0.0
59              0.0             0.0
60              0.0             0.0
61              0.0             0.0
62              0.0             0.0
63              0.0             0.0

[64 rows x 14 columns]
```

```
In [22]: # Obtain best classifier from grid search
         clf_best = gs.best_estimator_

In [23]: print('best C value =', clf_best.C)

best C value = 0.3


In [24]: print('best gamma value =', clf_best.gamma)

best gamma value = 100.0
```

In [25]:
```
plot_data(X_train_0, y_train_0)
plot_boundary(X_train_0, clf_best)
plt.title(f'C={clf_best.C:.4}, gamma={clf_best.gamma:.4}')
```

Out[25]: <matplotlib.text.Text at 0x108d4c6a0>




### 0.1.3 Spam classification

In [26]:
```
# Load spam classification training data
data_train = sio.loadmat('data/ml-ex6/spamTrain.mat')
# Load spam classification test data
data_test = sio.loadmat('data/ml-ex6/spamTest.mat')
```

In [27]:
```
# Sample feature vectors
X_train = data_train['X']
X_test = data_test['Xtest']

# Sample classes (0=non-spam, 1=spam)
y_train = data_train['y'].ravel()
y_test = data_test['ytest'].ravel()
```

In [28]:
```
# Train a linear SVC (running liblinear) using the
# same value for C as in in the original exercise
# (see ex6_spam.m)
```

```
clf = LinearSVC(C=0.1)
clf.fit(X_train, y_train)
```

```
Out[28]: LinearSVC(C=0.1, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
```

```
In [29]: # Training score
clf.score(X_train, y_train)
```

```
Out[29]: 0.99975000000000003
```

```
In [30]: # Test score
clf.score(X_test, y_test)
```

```
Out[30]: 0.99199999999999999
```

```
In [31]: vocab = {}

# Read provided vocabulary from vocab.txt file and add
# content to vocab, converting 1-based to 0-based index
with open('data/ml-ex6/vocab.txt') as vocab_file:
    for row in csv.reader(vocab_file, delimiter='\t'):
        word = row[1]
        index = int(row[0]) - 1
        vocab[word] = index
```

```
In [32]: def read_file(file):
    with open(file) as f:
        return f.read()

# Read non-spam email samples
email_1 = read_file('data/ml-ex6/emailSample1.txt')
email_2 = read_file('data/ml-ex6/emailSample2.txt')

# Read spam email samples
spam_1 = read_file('data/ml-ex6/spamSample1.txt')
spam_2 = read_file('data/ml-ex6/spamSample2.txt')
```

```
In [33]: # Print first 200 characters of a non-spam sample
print(email_1[:200], '...')
```

```
> Anyone knows how much it costs to host a web portal ?
>
Well, it depends on how many visitors you're expecting.
This can be anywhere from less than 10 bucks a month to a couple of $100.
You should  ...
```

```
In [34]: # Print first 200 characters of a spam sample
         print(spam_1[:200], '...')
```

```
Do You Want To Make $1000 Or More Per Week?



If you are a motivated and qualified individual - I
will personally demonstrate to you a system that will
make you $1,000 per week or more! This is NO ...
```

**Email pre-processing** In the following, we reuse email pre-processing logic from the original course exercise and use oct2py for calling Octave functions from Python. The called function for email pre-processing is `processEmail` which is located in directory `func/ml-ex6/`. It is a modified version of the original function and returns pre-processed words instead of their index in the dictionary (to avoid sharing Octave code that must be written during assignments). For a more detailed description of `processEmail`, see ex6.pdf, section 2.1.

```
In [35]: # Add directory func/ml-ex6 to Octave path.
         # It contains file/function processEmail.m
         octave.addpath('./func/ml-ex6');
```

```
In [36]: # Pre-process emails with provided Octave code.
         proc_email_1 = octave.processEmail(email_1)
         proc_email_2 = octave.processEmail(email_2)
         proc_spam_1 = octave.processEmail(spam_1)
         proc_spam_1 = octave.processEmail(spam_1)
```

```
warning: implicit conversion from numeric to char
warning: implicit conversion from numeric to char
warning: implicit conversion from numeric to char
warning: implicit conversion from numeric to char
```

```
In [37]: # Pre-processed non-spam sample
         proc_email_1
```

```
Out[37]: array(['anyon     ', 'know      ', 'how       ', 'much      ',
                 'it        ', 'cost      ', 'to        ', 'host      ',
                 'a         ', 'web       ', 'portal    ', 'well      ',
                 'it        ', 'depend    ', 'on        ', 'how       ',
                 'mani      ', 'visitor   ', 'you       ', 're        ',
                 'expect    ', 'thi       ', 'can       ', 'be        ',
                 'anywher   ', 'from      ', 'less      ', 'than      ',
                 'number    ', 'buck      ', 'a         ', 'month     ',
                 'to        ', 'a         ', 'coupl     ', 'of        ',
                 'dollarnumb', 'you       ', 'should    ', 'checkout  ',
                 'httpaddr  ', 'or        ', 'perhap    ', 'amazon    ',
```

```
       'ecnumb    ', 'if        ', 'your      ', 'run       ',
       'someth    ', 'big       ', 'to        ', 'unsubscrib',
       'yourself  ', 'from      ', 'thi       ', 'mail      ',
       'list      ', 'send      ', 'an        ', 'email     ',
       'to        ', 'emailaddr '],
      dtype='<U10')
```

In [38]:
```
def words_to_features(words, vocab):
    '''Creates a feature vector from a list of words.

    The length of the returned feature vector is equal to
    size of the vocabulary. The feature vector element i is
    1 if the i-th word in the vocabulary is in the words list,
    0 otherwise.
    '''

    features = np.zeros(len(vocab))
    for word in words:
        idx = vocab.get(word.strip())
        if idx:
            features[idx] = 1
    return features
```

In [39]:
```
# Convert pre-processed samples to feature vectors
X_samples = np.vstack([words_to_features(proc_email_1, vocab),
                       words_to_features(proc_email_2, vocab),
                       words_to_features(proc_spam_1, vocab),
                       words_to_features(proc_spam_1, vocab)])
```

In [40]:
```
# Predict the classes of samples (should be
# [0, 0, 1, 1] where 0=non-spam and 1=spam)
clf.predict(X_samples)
```

Out[40]:
```
array([0, 0, 1, 1], dtype=uint8)
```