COMP 540 Statistical Machine Learning HW4

Chuanwenjie Wei, Yiqing Lu

March 6, 2017

1 Kernelizing k-nearest neighbors

• **The classical KNN:** when a new data point x is introduced, we first need to compute the Euclidean distance between x and $x^{(i)}$ for any i :

$$d\left(x, x^{(i)}\right) \quad = \quad ||x - x^{(i)}||_2$$

which can also be represented as a dot product:

$$d\left(x, x^{(i)}\right) \quad = \quad \sqrt{< x, x > + < x^{(i)}, x^{(i)} > -2 < x, x^{(i)} >}$$

where $<>$ denotes the inner product.

Then we are supposed to choose k closest distance to predict the label of new x.

• **Kernel View:** From the perspective of kernel, we meant to find a function ϕ to map all the points to a higher dimension space firstly.

But here is the kernel trick comes in, which means in fact, there exists a function (the Kernel Function k satisfied Mercer Condition) to compute the inner product in the higher-dimensional space without first transforming the vectors into the higher-dimensional space.

So for the kenelized KNN, we have:

$$d\left(\phi\left(x\right), \phi\left(x^{(i)}\right)\right) \quad = \quad \sqrt{< \phi\left(x\right), \phi\left(x\right) > + < \phi\left(x^{(i)}\right), \phi\left(x^{(i)}\right) > -2 < \phi\left(x\right), \phi\left(x^{(i)}\right) >}$$

$$= \quad \sqrt{k\left(x, x\right) + k\left(x^{(i)}, x^{(i)}\right) - 2k\left(x, x^{(i)}\right)}$$

2 Constructing kernels

It's obvious that these three are symmetric, so we only need to show that they are positive semi-definite.

• Let $K1, K2$ denotes the kernel matrix corresponding to kernel function k_1, k_2

Then for any z, we have:

$$z^T K_{new} z = c\left(z^T K_1 z\right) \geq 0$$

when $c \geq 0$

So the new kernel $ck_1\left(x, x^{'}\right)$ is valid when $c \geq 0$.

•We know that if k_1 is a valid kernel, there exists ϕsuch that:

$$k_1\left(x, x^{'}\right) = <\phi(x), \phi(x^{'})>$$

where <> denotes an inner product

So we can rewrite k_{new} as:

$$k_{new}\left(x, x^{'}\right) = f\left(x\right) <\phi(x), \phi(x^{'})> f\left(x^{'}\right)$$

$$= <f\left(x\right)\phi(x), f\left(x^{'}\right)\phi(x^{'})>$$

$$= <\phi_{new}(x), \phi_{new}(x^{'})>$$

Therefore, k_{new} can be represented as an inner product on some Hilbert Space.

So $k_{new}\left(x, x^{'}\right) = f\left(x\right) k_1\left(x, x^{'}\right) f\left(x^{'}\right)$ is positive semi-definite.

•For any z, we have:

$$z^T K_{new} z = z^T\left(K_1 + K_2\right) z$$

$$= z^T K_1 z + z^T K_2 z$$

Since $k_1,\ k_2$ are all valid kernels, $z^T K_{new} z \geq 0$

So $k_1\left(x, x^{'}\right) + k_2\left(x, x^{'}\right)$ is also a valid kernel.

3 Fitting an SVM classifier by hand

3.1 A vector parallel to the optimal vector

Let $\theta = (a, b, c)^T$,

Since θ is perpendicular to the decision boundry between the two points in the three-dimensional space, and the decision boundry is on the perpendicular bisector of these two points,

Soθ is parallel to the vector connecting these two points:

$$k\,(a, b, c) \quad = \quad (1, 2, 2) - (1, 0, 0)$$

Therefore, we can derive one of the parallel vector equals$(0, 2, 2)$

3.2 value of the margin

Since the margin is the distance from each support vector to the decision boundary, So

$$\gamma \quad = \quad \frac{1}{2}\sqrt{(1-1)^2 + (2-0)^2 + (2-0)^2} = \sqrt{2}$$

3.3 Solve for the θ

We know that $\theta = (a, b, c)^T$, and $\gamma = \sqrt{2}$.

So given that $\gamma = \frac{1}{||\theta||}$, we can derive:

$$\frac{1}{||\theta||} \quad = \quad \sqrt{2} \Longrightarrow a^2 + b^2 + c^2 = \frac{1}{2}$$

From the previous part, we know θ is parrallel to $(0, 2, 2)^T$, which is equivalent to:

$$a \quad = \quad 0,\ b = c$$

Combining these two equations, we get:

$$2b^2 \quad = \quad \frac{1}{2} \Longrightarrow b = \pm\frac{1}{2}$$

Considering the direction of vector, we let $b = \frac{1}{2}$ and therefore $\theta = \left(0, \frac{1}{2}, \frac{1}{2}\right)^T$

3.4 Solve for the intercept θ_0

We already know that

$$y^{(1)}\left(\theta^T\phi(x^{(1)}) + \theta_0\right) \quad \geq \quad 1$$

$$y^{(2)}\left(\theta^T\phi(x^{(2)}) + \theta_0\right) \quad \geq \quad 1$$

Plugging the results from the previous parts into these two equations:

$$-1 * \left(0 * 1 + \frac{1}{2} * 0 + \frac{1}{2} * 0 + \theta_0\right) \geq 1 \Longrightarrow \theta_0 \leq -1$$

$$1 * \left(0 * 1 + \frac{1}{2} * 2 + \frac{1}{2} * 2 + \theta_0\right) \geq 1 \Longrightarrow \theta_0 \geq -1$$

So

$$\theta_0 = -1$$

3.5 The equation for the decision boundary

Using the results above, we can derive the equation of the decision boundary:

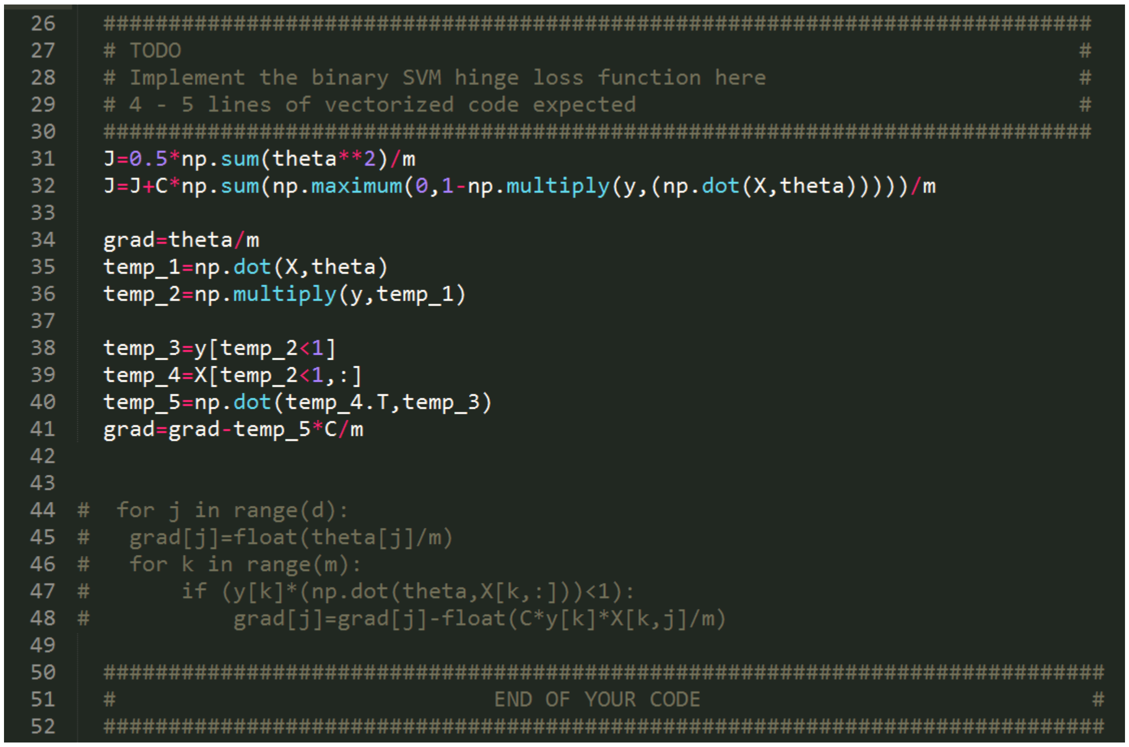
$$\theta^T \phi(x) + \theta_0 = 0 \Longrightarrow \left(0, \frac{1}{2}, \frac{1}{2}\right) * \left(1, \sqrt{2}x, x^2\right)^T - 1 = 0$$

$$\Longrightarrow \frac{1}{\sqrt{2}}x + \frac{1}{2}x^2 - 1 = 0$$

4 Support vector machines for binary classicationneighbors

4.1 Support vector machines

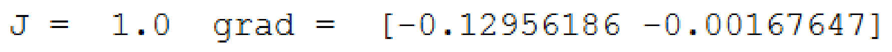
4.1A The hinge loss function and gradient



```
26    ##########################################################################
27    # TODO                                                                   #
28    # Implement the binary SVM hinge loss function here                      #
29    # 4 - 5 lines of vectorized code expected                                #
30    ##########################################################################
31    J=0.5*np.sum(theta**2)/m
32    J=J+C*np.sum(np.maximum(0,1-np.multiply(y,(np.dot(X,theta)))))/m
33
34    grad=theta/m
35    temp_1=np.dot(X,theta)
36    temp_2=np.multiply(y,temp_1)
37
38    temp_3=y[temp_2<1]
39    temp_4=X[temp_2<1,:]
40    temp_5=np.dot(temp_4.T,temp_3)
41    grad=grad-temp_5*C/m
42
43
44  #  for j in range(d):
45  #   grad[j]=float(theta[j]/m)
46  #   for k in range(m):
47  #       if (y[k]*(np.dot(theta,X[k,:]))<1):
48  #           grad[j]=grad[j]-float(C*y[k]*X[k,j]/m)
49
50    ###########################################################################
51    #                              END OF YOUR CODE                           #
52    ###########################################################################
```

Fig.1 The code of the hinge loss and gradient

The code of the hinge loss and gradient is shown in Fig.1. The code from line 31 to line 41 is the vectorized version, and the code from line 44 to line 48 is the naive version. The result is shown in Fig.2.



```
J =  1.0  grad =  [-0.12956186 -0.00167647]
```

Fig.2 The result of the hinge loss and gradient

4.1B Example dataset 1: impact of varying C

When $C = 1$, we can see that the accuracy on training data = 0.980392156863, and the decision boundary is shown in Fig.3.

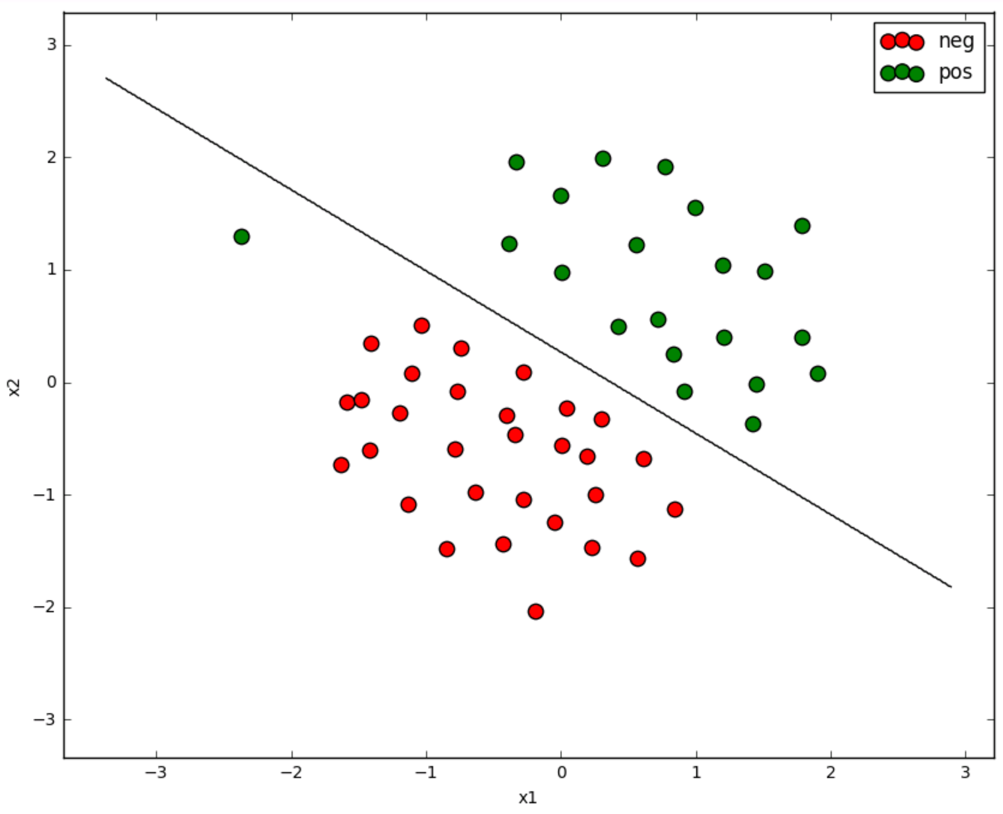


Fig.3 The result of the decision boundary when $C = 1$

When $C = 100$, we can see that the accuracy on training data = 1.0, and the decision boundary is shown in Fig.4.

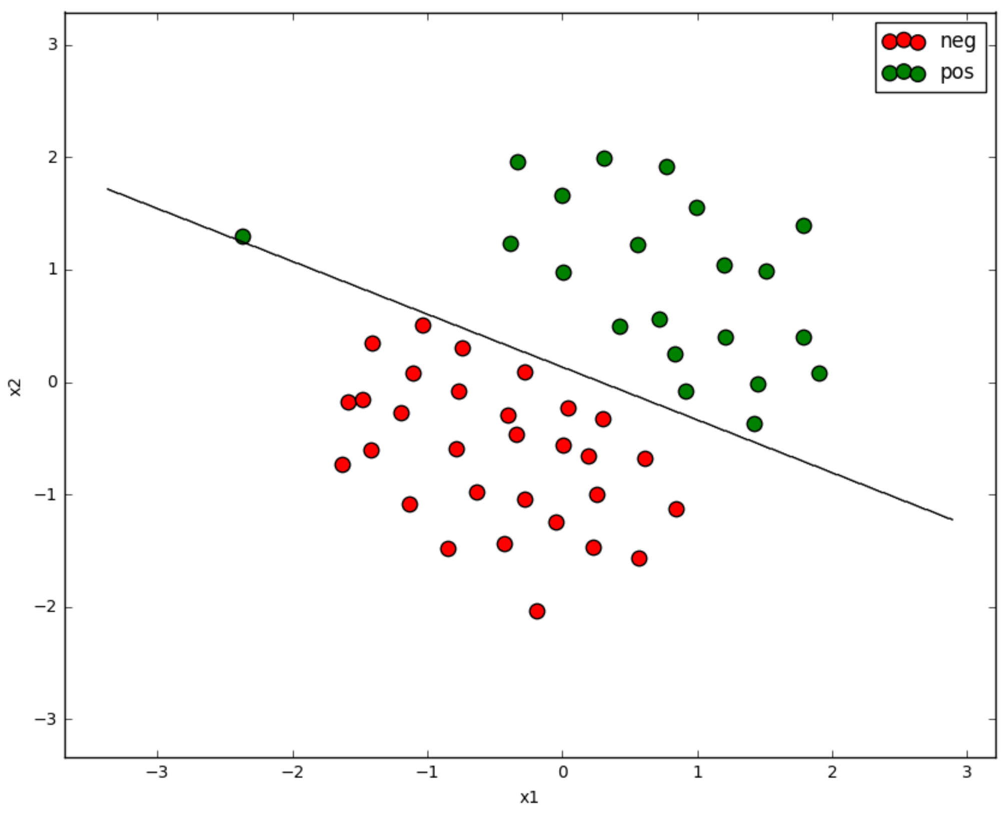
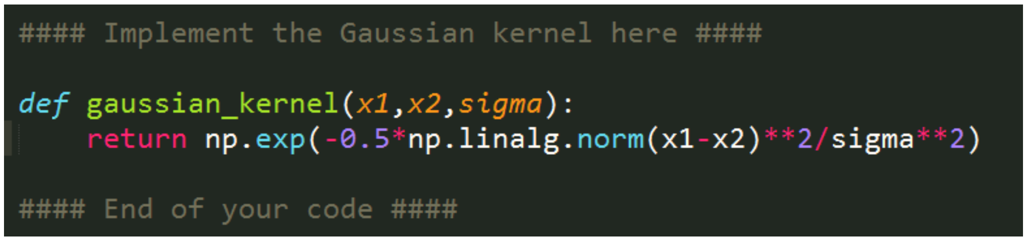


Fig.4 The result of the decision boundary when $C = 100$

From the result in Fig.3 and Fig.4, we can see that there is a point mismatching. It's mainly because of the small regularization strength. If a point is mismatching, the regularization part doesn't occupy to much on the loss function and gradient function. If we have a big value of C, in this case C=100, the regularization part occupies a lot on both the loss function and gradient function, so it's unlikely to appear a mismatching point.

4.1C Gaussian kernel

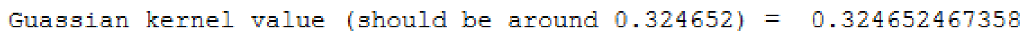
The code of gaussian kernal is shown in Fig.5.



```
#### Implement the Gaussian kernel here ####

def gaussian_kernel(x1,x2,sigma):
    return np.exp(-0.5*np.linalg.norm(x1-x2)**2/sigma**2)

#### End of your code ####
```

Fig.5 The code of gaussian kernel

The result of the gaussian kernal for $x_1 = (1, 2, 1)^T$, $x_2 = (0, 4, -1)^T$ is shown in Fig.6.



```
Guassian kernel value (should be around 0.324652) =  0.324652467358
```

Fig.6 The result of gaussian kernel

• **Example dataset 2: learning non-linear boundaries** The result of the decision boundary when $C = 1$ and $\sigma = 0.02$ is shown in Fig.7.

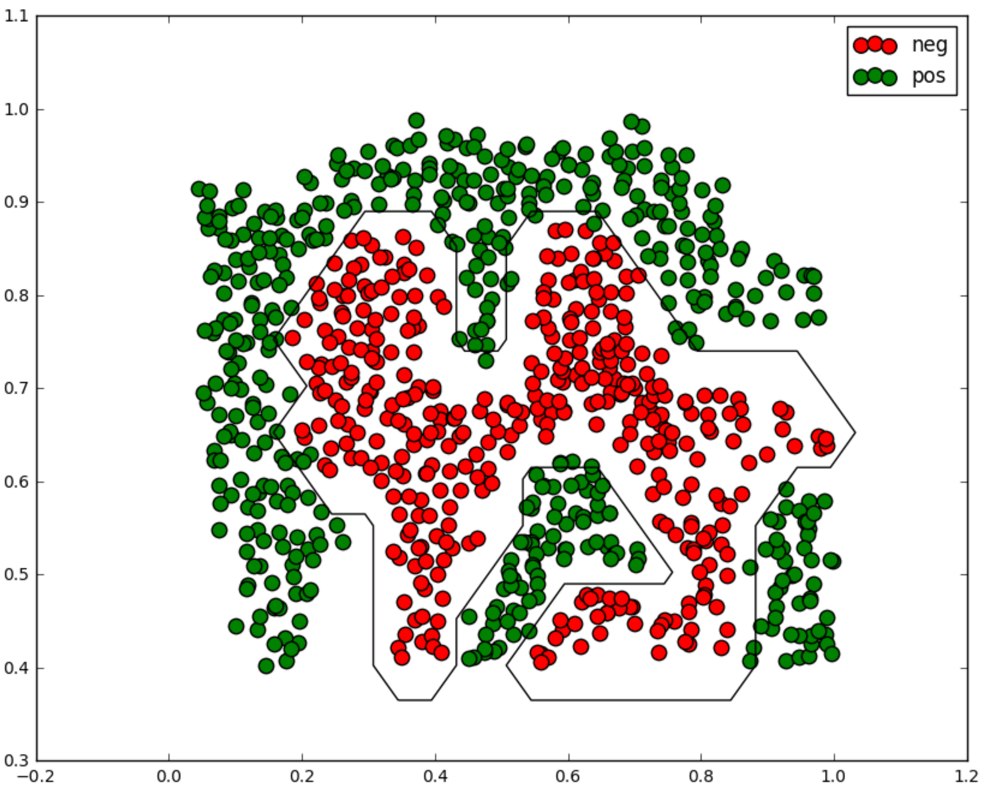


Fig.7 The result of the decision boundary when $C = 1$ and $\sigma = 0.02$

From the result, we can see that the number mismatching points is very small the gaussian kernel fits well on this data set.

4.2 Example dataset 3: selecting hyper parameters for SVMs

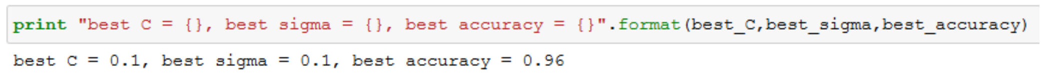
The code of selecting hyper parameters is as follows.

```
best_accuracy=0
best_C=Cvals[0]
best_sigma=sigma_vals[0]
for C in Cvals:
    for sigma_val in sigma_vals:
        K=np.array([utils.gaussian_kernel(x1,x2,sigma_val)
        for x1 in X for x2 in X]).reshape(X.shape[0],X.shape[0])
        scaler=preprocessing.StandardScaler().fit(K)
```

```
scaleK=scaler.transform(K)
KK = np.vstack([np.ones((scaleK.shape[0],)),scaleK.T]).T
Kval = np.array([utils.gaussian_kernel(x1,x2,sigma_val)
for x1 in Xval for x2 in X]).reshape(Xval.shape[0],X.shape[0])
scaler = preprocessing.StandardScaler().fit(Kval)
scaleK = scaler.transform(Kval)
KKval = np.vstack([np.ones((scaleK.shape[0],)),scaleK.T]).T
svm = LinearSVM_twoclass()
svm.theta = np.zeros((KK.shape[1],))
svm.train(KK,yy,learning_rate=1e-4,reg=C,num_iters=20000,verbose=True,
step=5000)
yyval_pred = svm.predict(KKval) accuracy=np.mean(yyval==yyval_pred)
if best_accuracy<=accuracy:
    best_accuracy=accuracy
    best_C=C
    best_sigma=sigma_val
    print "When C = {} and sigma = {}, vali-
    dation accuracy = {}".format(C,sigma_val,accuracy)
    print '='*70
```

All the result is shown in binary_sum.ipunb.

And we print the best C and best sigma shown in Fig.8.

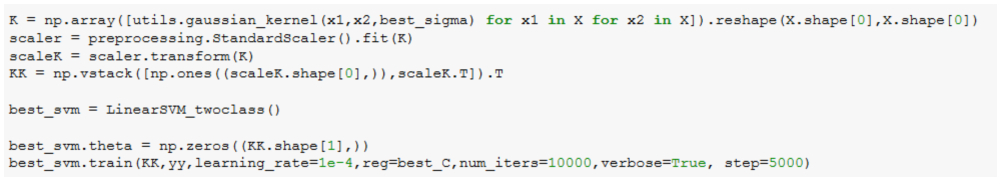


```
print "best C = {}, best sigma = {}, best accuracy = {}".format(best_C,best_sigma,best_accuracy)
```

```
best C = 0.1, best sigma = 0.1, best accuracy = 0.96
```

Fig.8 The result of the best C and best sigma

Then we calculate the best SVM. The code is shown in Fig.9.



```
K = np.array([utils.gaussian_kernel(x1,x2,best_sigma) for x1 in X for x2 in X]).reshape(X.shape[0],X.shape[0])
scaler = preprocessing.StandardScaler().fit(K)
scaleK = scaler.transform(K)
KK = np.vstack([np.ones((scaleK.shape[0],)),scaleK.T]).T

best_svm = LinearSVM_twoclass()

best_svm.theta = np.zeros((KK.shape[1],))
best_svm.train(KK,yy,learning_rate=1e-4,reg=best_C,num_iters=10000,verbose=True, step=5000)
```

Fig.9 The code of calculating the best SVM

The decision boundary of the best SVM is shown in Fig.10.

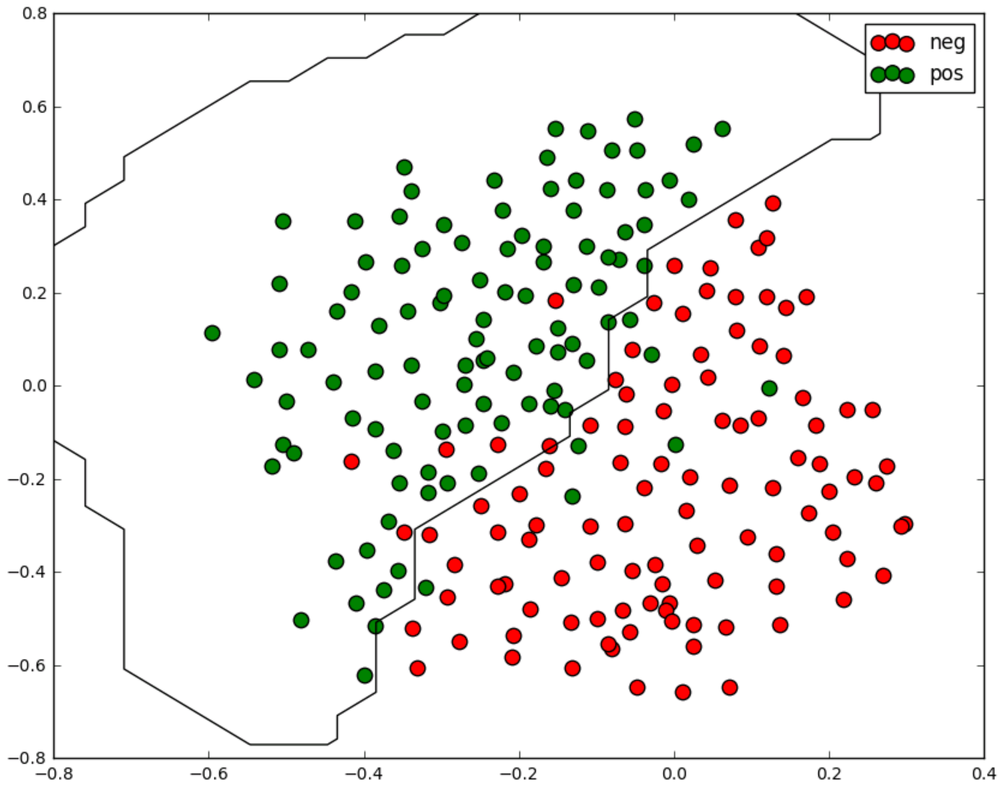


Fig.10 The decision boundary of the best SVM

4.3 Example dataset 3: selecting hyper parameters for SVMs

The code of choosing the best hyper parameters is shown as follows.

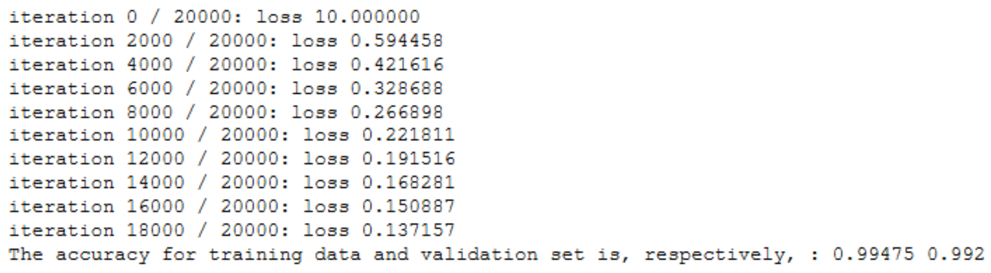
```
Cvals = [10,30,50,100]
iterationvals = [2000,5000,10000]
learning_ratevals = [1e-3,1e-2,1e-1]
best_C = 0
best_learning_rate = 0
best_iteration = 0
best_val_acc = 0
for C in Cvals:
    for iteration in iterationvals:
        for learning_rate in learning_ratevals:
```

```
        print "training with C: {}, num_iters: {},
        learning_rate: {}".format(C, iteration, learn-
        ing_rate)
        svm = LinearSVM_twoclass()
        svm.theta = np.zeros((X.shape[1],))
        svm.train(X, yy, learning_rate, reg=C, num_iters=iteration,
        verbose=True, step=1000)
        val_acc = np.mean(svm.predict(X_test)
        == yy_test)
        print "When C = {}, num_iters = {},
        learning_rate = {}, validation accuracy
        = {}".format(C, iteration, learning_rate,
        val_acc)
        print("-"*70)
        if val_acc >= best_val_acc: best_C =
        C
          best_iteration = iteration
          best_learning_rate = learning_rate
          best_val_acc=val_acc

print "So the best paramater set we derive is:", best_C, best_iteration,
best_learning_rate
```

Becaust X is the one-hot code, we don't use the Gaussian kernel to transform the data set since the mean of X is near 0.

From the result in the file 'svm_spam_resrlt.html' (The code runs slow so we devide them into several files to operate and combine all the results in the html), we can see that the best accuracy could be gotten when learning rate=0.001, regularization length=10 and iteration number is 10000.

And we add the iteration number to 20000. The result is shown in Fig.11.



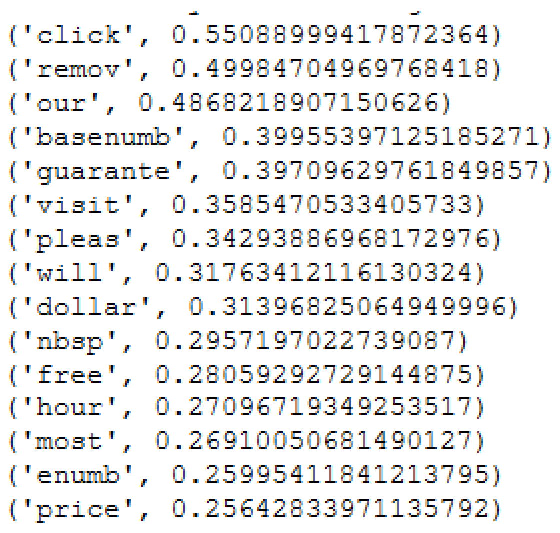
```
iteration 0 / 20000: loss 10.000000
iteration 2000 / 20000: loss 0.594458
iteration 4000 / 20000: loss 0.421616
iteration 6000 / 20000: loss 0.328688
iteration 8000 / 20000: loss 0.266898
iteration 10000 / 20000: loss 0.221811
iteration 12000 / 20000: loss 0.191516
iteration 14000 / 20000: loss 0.168281
iteration 16000 / 20000: loss 0.150887
iteration 18000 / 20000: loss 0.137157
The accuracy for training data and validation set is, respectively, : 0.99475 0.992
```

Fig.11 The result of the accuracy with the best parameter and iteration=20000

Then we use the code to sort the words.

```
words, inv_words = utils.get_vocab_dict()
theta = svm.theta
np.argsort(theta)[::-1]
for i in np.argsort(theta)[::-1]:
    print(words[i + 1], theta[i])
```

The result of the top 15 words are



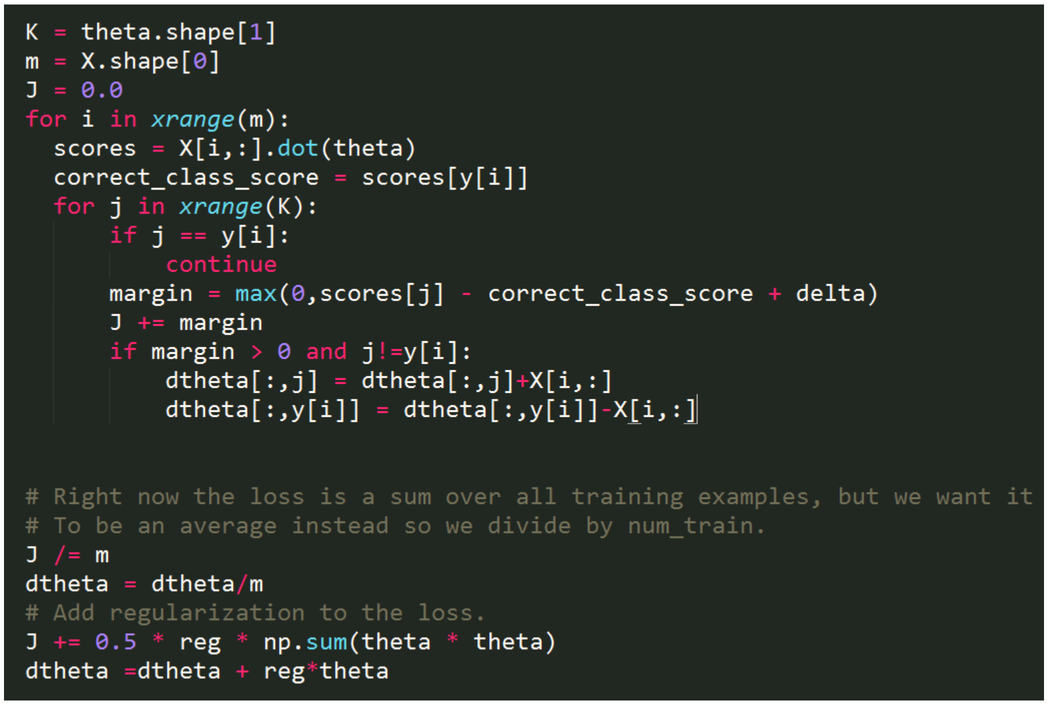
```
('click', 0.55088999417872364)
('remov', 0.49984704969768418)
('our', 0.4868218907150626)
('basenumb', 0.39955397125185271)
('guarante', 0.39709629761849857)
('visit', 0.3585470533405733)
('pleas', 0.34293886968172976)
('will', 0.31763412116130324)
('dollar', 0.31396825064949996)
('nbsp', 0.2957197022739087)
('free', 0.28059292729144875)
('hour', 0.27096719349253517)
('most', 0.26910050681490127)
('enumb', 0.25995411841213795)
('price', 0.25642833971135792)
```

The whole result is shown in the file 'svm_spam_resrlt.html'.

5 Support vector machines for multi-class classication

5.1 Loss and gradient function for multi-class SVM (naive version)

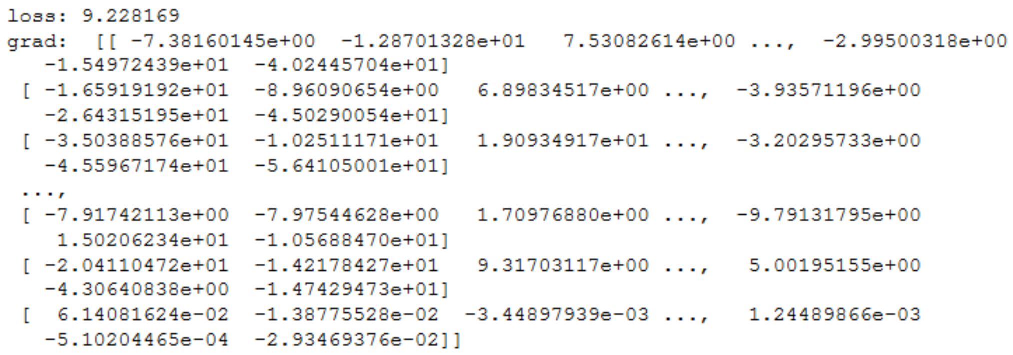
The code of calculating loss and gradient function for multi-class SVM with naive version is shown in Fig.12.



```python
K = theta.shape[1]
m = X.shape[0]
J = 0.0
for i in xrange(m):
  scores = X[i,:].dot(theta)
  correct_class_score = scores[y[i]]
  for j in xrange(K):
      if j == y[i]:
          continue
      margin = max(0,scores[j] - correct_class_score + delta)
      J += margin
      if margin > 0 and j!=y[i]:
          dtheta[:,j] = dtheta[:,j]+X[i,:]
          dtheta[:,y[i]] = dtheta[:,y[i]]-X[i,:]


# Right now the loss is a sum over all training examples, but we want it
# To be an average instead so we divide by num_train.
J /= m
dtheta = dtheta/m
# Add regularization to the loss.
J += 0.5 * reg * np.sum(theta * theta)
dtheta =dtheta + reg*theta
```

Fig.12 The code of loss and gradient function for multi-class SVM with naive version

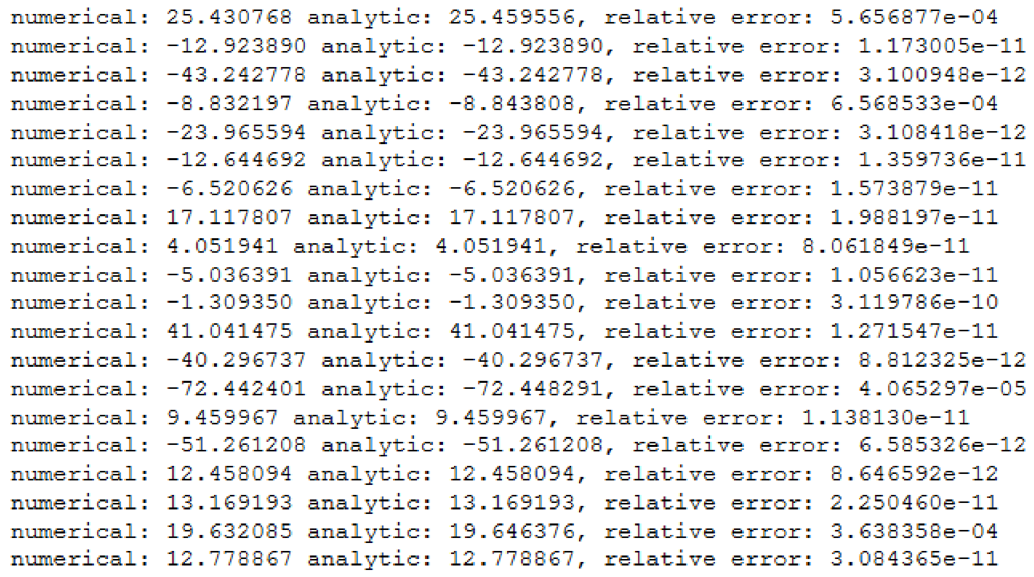
The result of loss and gradient function for multi-class SVM with train data set is shown in Fig.13.



```
loss: 9.228169
grad:  [[ -7.38160145e+00  -1.28701328e+01   7.53082614e+00 ...,  -2.99500318e+00
   -1.54972439e+01  -4.02445704e+01]
 [ -1.65919192e+01  -8.96090654e+00   6.89834517e+00 ...,  -3.93571196e+00
   -2.64315195e+01  -4.50290054e+01]
 [ -3.50388576e+01  -1.02511171e+01   1.90934917e+01 ...,  -3.20295733e+00
   -4.55967174e+01  -5.64105001e+01]
 ...,
 [ -7.91742113e+00  -7.97544628e+00   1.70976880e+00 ...,  -9.79131795e+00
    1.50206234e+01  -1.05688470e+01]
 [ -2.04110472e+01  -1.42178427e+01   9.31703117e+00 ...,   5.00195155e+00
   -4.30640838e+00  -1.47429473e+01]
 [  6.14081624e-02  -1.38775528e-02  -3.44897939e-03 ...,   1.24489866e-03
   -5.10204465e-04  -2.93469376e-02]]
```

Fig.13 The result of loss and gradient function for multi-class SVM

The error between the numeric estimate and the computed gradient is shown in Fig.14.



```
numerical: 25.430768 analytic: 25.459556, relative error: 5.656877e-04
numerical: -12.923890 analytic: -12.923890, relative error: 1.173005e-11
numerical: -43.242778 analytic: -43.242778, relative error: 3.100948e-12
numerical: -8.832197 analytic: -8.843808, relative error: 6.568533e-04
numerical: -23.965594 analytic: -23.965594, relative error: 3.108418e-12
numerical: -12.644692 analytic: -12.644692, relative error: 1.359736e-11
numerical: -6.520626 analytic: -6.520626, relative error: 1.573879e-11
numerical: 17.117807 analytic: 17.117807, relative error: 1.988197e-11
numerical: 4.051941 analytic: 4.051941, relative error: 8.061849e-11
numerical: -5.036391 analytic: -5.036391, relative error: 1.056623e-11
numerical: -1.309350 analytic: -1.309350, relative error: 3.119786e-10
numerical: 41.041475 analytic: 41.041475, relative error: 1.271547e-11
numerical: -40.296737 analytic: -40.296737, relative error: 8.812325e-12
numerical: -72.442401 analytic: -72.448291, relative error: 4.065297e-05
numerical: 9.459967 analytic: 9.459967, relative error: 1.138130e-11
numerical: -51.261208 analytic: -51.261208, relative error: 6.585326e-12
numerical: 12.458094 analytic: 12.458094, relative error: 8.646592e-12
numerical: 13.169193 analytic: 13.169193, relative error: 2.250460e-11
numerical: 19.632085 analytic: 19.646376, relative error: 3.638358e-04
numerical: 12.778867 analytic: 12.778867, relative error: 3.084365e-11
```

Fig.14 The error between the numeric estimate and the computed gradient

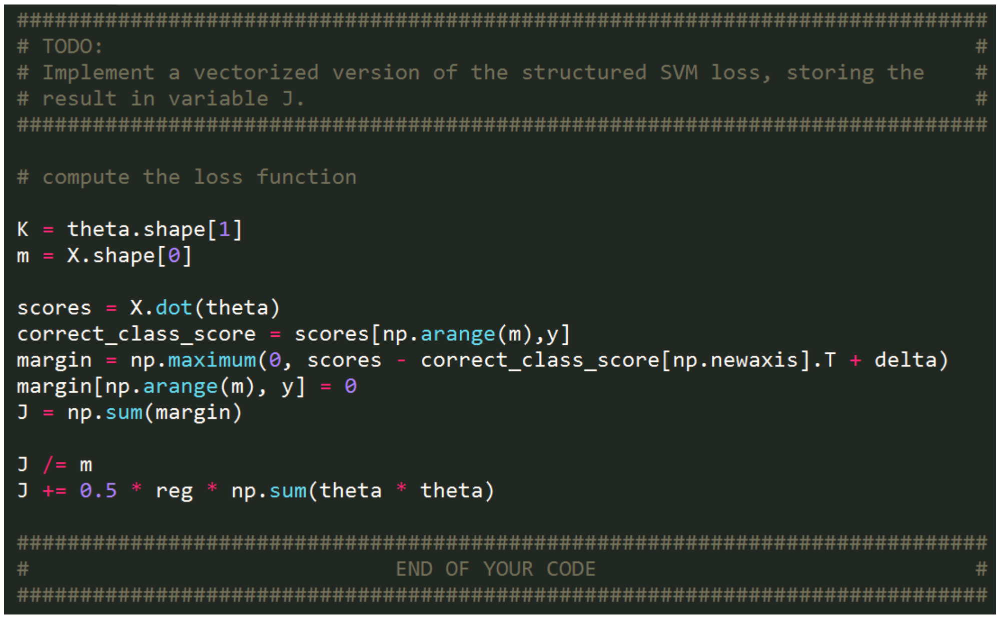
From the Fig.13, we can see that there is a small relative error between the numeric estimate and the computed gradient. It's up to 6.568533e-04. However, It is possible that once in a while a dimension in the gradient check will not match exactly. I think it's mainly because of the expression:

$$L = \sum_{j \neq y^{(i)}} max\left(\; 0, \quad \theta^{(j)^T} x^{(i)} - \theta^{y^{(j)^T}} x^{(i)} + \Delta \; \right)$$

The numeric estimate is based on the difference of the loss function. So the gradient of the expression is not continuous. From the expression, we can see that when $j = y^{(i)}$, the gradient should change sharply. So maybe in this dimension, the gradient we compute is different from the different equation result.

5.2 Loss and gradient function for multi-class SVM (vectorized version)

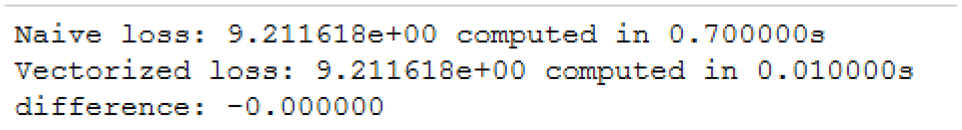
The code of calculating loss and gradient function for multi-class SVM with vectorized version is shown in Fig.15.



```
#############################################################################
# TODO:                                                                     #
# Implement a vectorized version of the structured SVM loss, storing the    #
# result in variable J.                                                     #
#############################################################################

# compute the loss function

K = theta.shape[1]
m = X.shape[0]

scores = X.dot(theta)
correct_class_score = scores[np.arange(m),y]
margin = np.maximum(0, scores - correct_class_score[np.newaxis].T + delta)
margin[np.arange(m), y] = 0
J = np.sum(margin)

J /= m
J += 0.5 * reg * np.sum(theta * theta)

#############################################################################
#                              END OF YOUR CODE                             #
#############################################################################
```

Fig.15 The code of loss and gradient function for multi-class SVM with vectorized version

The difference between the naive loss and the vectorized loss is shown in Fig.16.



```
Naive loss: 9.211618e+00 computed in 0.700000s
Vectorized loss: 9.211618e+00 computed in 0.010000s
difference: -0.000000
```

Fig.16 The difference between the naive loss and the vectorized loss

From Fig.15, we can see that the vectorized version we computed is correct.

The relationship between the loss value and the iteration number is shown in Fig.17.

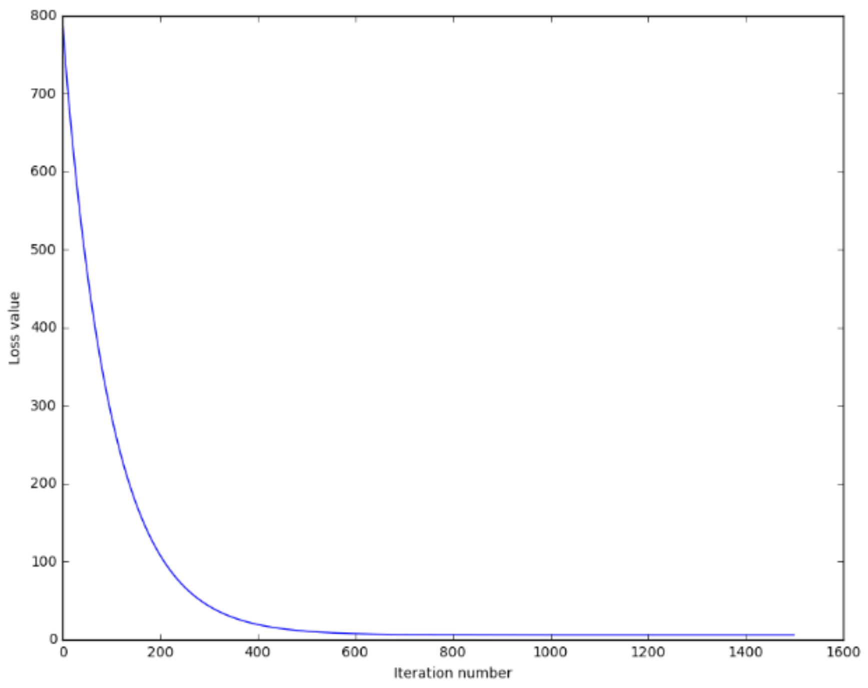
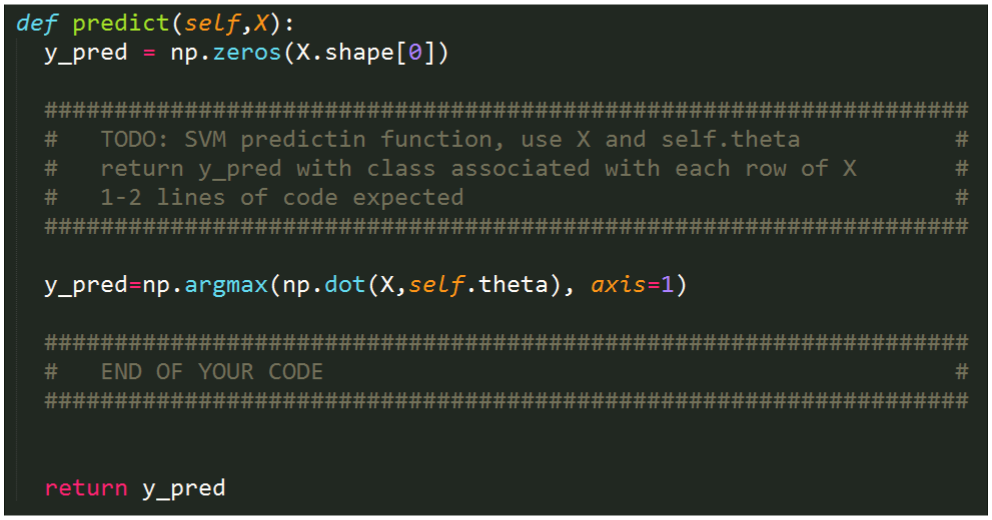


Fig.17 The relationship between the loss value and the iteration number

5.3 Prediction function for multi-class SVM

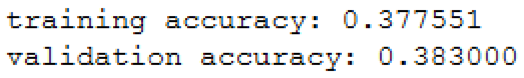
The code of the prediction function is shown in Fig.18.



```
def predict(self,X):
    y_pred = np.zeros(X.shape[0])

    ###################################################################
    #   TODO: SVM predictin function, use X and self.theta            #
    #   return y_pred with class associated with each row of X        #
    #   1-2 lines of code expected                                    #
    ###################################################################

    y_pred=np.argmax(np.dot(X,self.theta), axis=1)

    ###################################################################
    #   END OF YOUR CODE                                              #
    ###################################################################


    return y_pred
```

Fig.18 The code of the prediction function

The result of the training accuracy and the validation accuracy is shown in Fig.19.



```
training accuracy: 0.377551
validation accuracy: 0.383000
```

Fig.19 The training accuracy and the validation accuracy

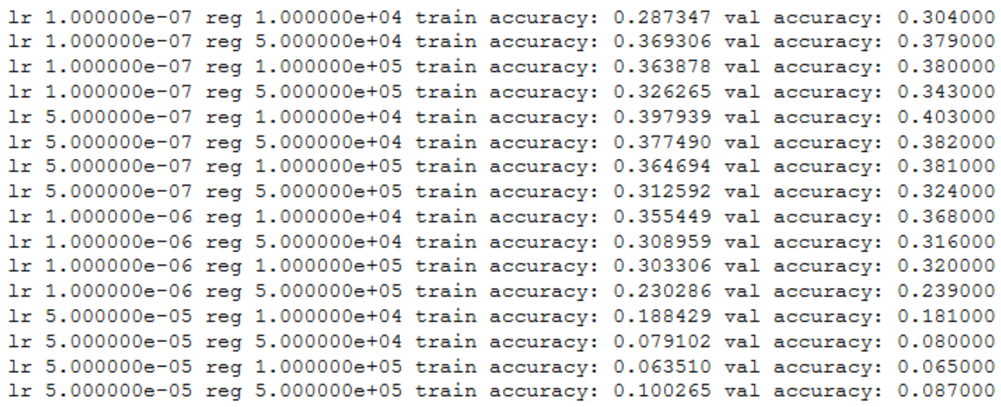
The result is as about 37-39% as expected.

5.4 Tuning hyper parameters for training a multi-class SVM

The code of selecting hyper parameters is as follows. We use a small number of iterations, in this case = 500 at first.

```
best_lamda=learning_rates[0]
best_reg=regularization_strengths[0]
for lamda in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        loss_hist=svm.train(X_train, y_train, lamda, reg,
        num_iters=500, verbose=True, step=100)
        y_val_pred = svm.predict(X_val)
        accuracy_val=np.mean(y_val == y_val_pred)
        y_train_pred=svm.predict(X_train)
        accuracy_train=np.mean(y_train == y_train_pred)
        results[(lamda, reg)] = (accuracy_train, accuracy_val)
        if accuracy_val>best_val:
            best_lamda=lamda
            best_reg=reg
            best_val=accuracy_val
            best_svm=svm
            print "When learning rate = {}, regular-
            ication length = {}, validation accuracy
            = {}".format(lamda, reg, accuracy_val)
            print("-"*70)
```

The result of the validation accuracy with different value of regularization length and learning rate is shown in Fig.20.

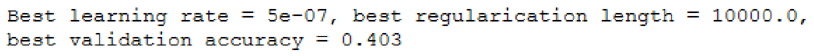


```
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.287347 val accuracy: 0.304000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.369306 val accuracy: 0.379000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.363878 val accuracy: 0.380000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.326265 val accuracy: 0.343000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.397939 val accuracy: 0.403000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.377490 val accuracy: 0.382000
lr 5.000000e-07 reg 1.000000e+05 train accuracy: 0.364694 val accuracy: 0.381000
lr 5.000000e-07 reg 5.000000e+05 train accuracy: 0.312592 val accuracy: 0.324000
lr 1.000000e-06 reg 1.000000e+04 train accuracy: 0.355449 val accuracy: 0.368000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.308959 val accuracy: 0.316000
lr 1.000000e-06 reg 1.000000e+05 train accuracy: 0.303306 val accuracy: 0.320000
lr 1.000000e-06 reg 5.000000e+05 train accuracy: 0.230286 val accuracy: 0.239000
lr 5.000000e-05 reg 1.000000e+04 train accuracy: 0.188429 val accuracy: 0.181000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.079102 val accuracy: 0.080000
lr 5.000000e-05 reg 1.000000e+05 train accuracy: 0.063510 val accuracy: 0.065000
lr 5.000000e-05 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
```

Fig.20 The validation accuracy with different value of regularization length and learning rate

And we get the best parameter shown in Fig.21.

```
print "Best learning rate = {}, best regularization length = {}, best
validation accuracy = {}".format(best_lamda, best_reg, best_val)
```



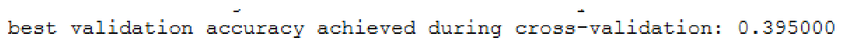
```
Best learning rate = 5e-07, best regularication length = 10000.0,
best validation accuracy = 0.403
```

Fig.21 The best parameter of the data set

Then we change the iteration number to 2000.

```
best_svm = LinearSVM()
best_loss_hist = best_svm.train(X_train, y_train, best_lamda,
best_reg, num_iters=2000, verbose=True, step=200)
y_val_pred = best_svm.predict(X_val)
best_val_cal=np.mean(y_val == y_val_pred)
```

And we get the best validation accuracy as shown in Fig.22.



```
best validation accuracy achieved during cross-validation: 0.395000
```

Fig.22 The best validation accuracy

The result of the training accuracy and the validation training accuracy with respect to the learning rate and the regularizaion length is shown in Fig.23.

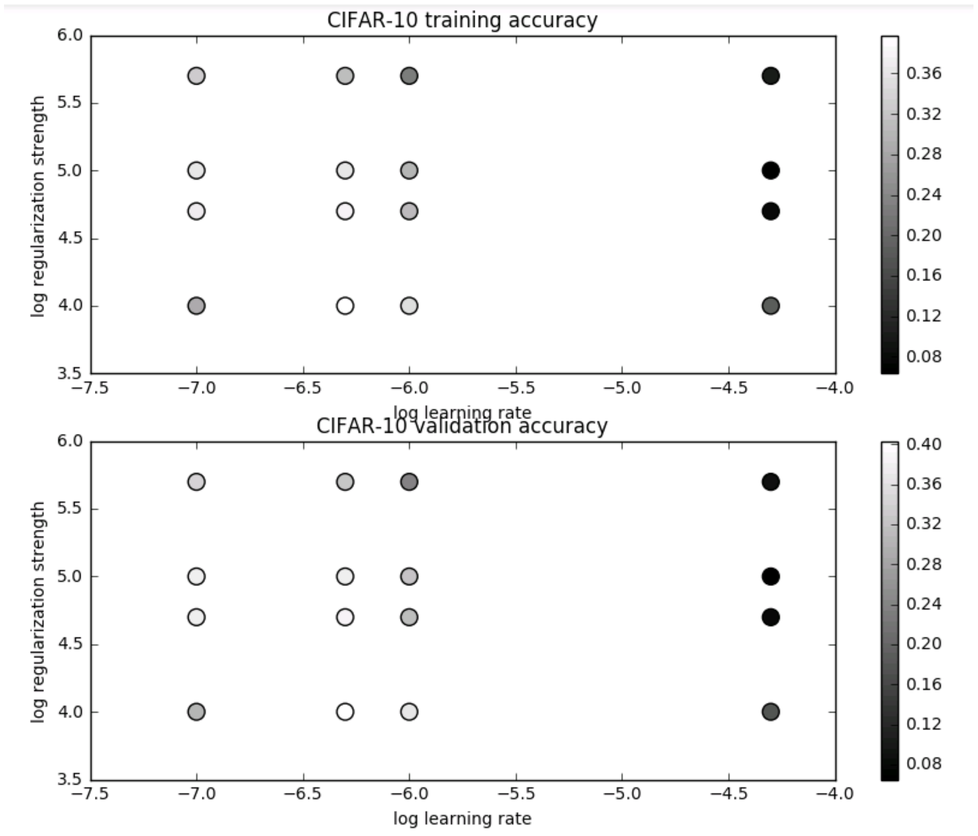
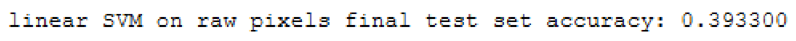


Fig.23 The result of the training accuracy and the validation training accuracy with respect to the learning rate and the regularizaion length

The result of test accuracy is shown in Fig.24.



```
linear SVM on raw pixels final test set accuracy: 0.393300
```

Fig.24 The result of the test accuracy

The visualization of learned weights for each class is shown in Fig.25.

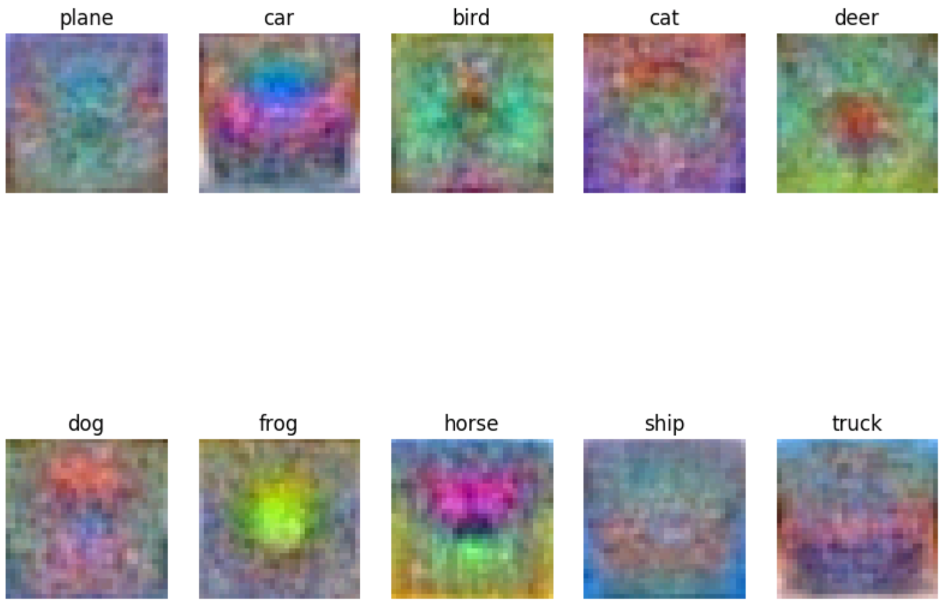


Fig.25 The visualization the learned weights for each class computed by multi-class SVM

5.5 Comparing the performance of multi-class SVM and softmax regression

Multi-class SVM takes a longer time than softmax regression. The test set accuracy we get from multi-class SVM is 0.393300, and the test set accuracy we get from softmax is 0.390800. So the multi-class SVM and the softmax have the similar performance. In this case, the performance of multi-class SVM is a little better than softmax.

The visualization of learned weights for each class computed by softmax is shown in Fig.26.

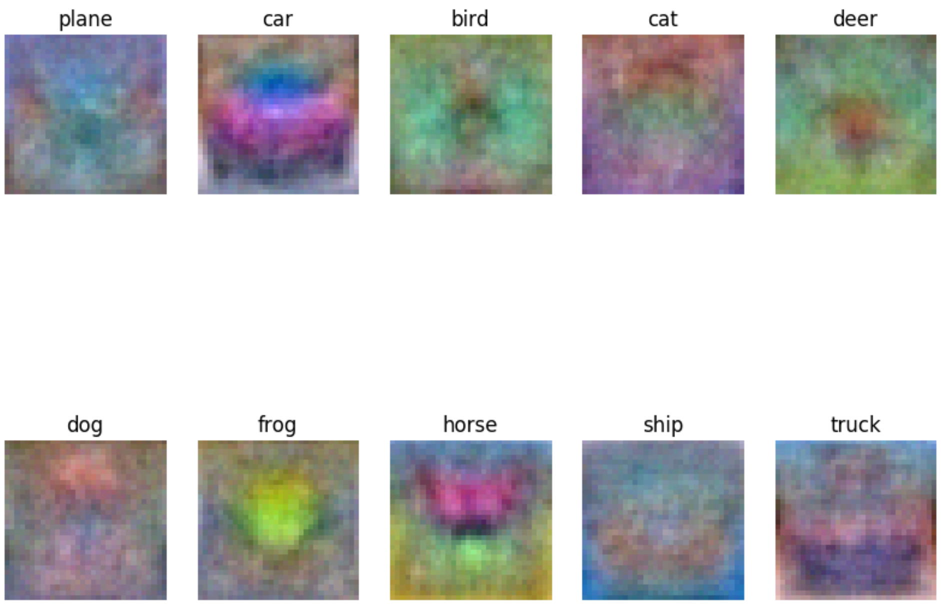


Fig.26 The visualization the learned weights for each class computed by softmax

From the Fig.24 and Fig.25, we can see that Fig.25 is smoother than the result in Fig.24. It's mainly because the result of multi-class SVM is based on the decision boundary. The points in the same decision boundary has the same label, and often the points of the same decision boundary gather in a certain area, so the result of the multi-class SVM looks discrete. The result of softmax, in comparison, is based on the the label of the largest probability. The label we predict is based on the maximum possibility we get from all the labels, so the points in a certain area is less likely to gather together. So the visualization computed by softmax is much smoother than the visualization computed by multi-class SVM.