CS534 Machine Learning Homework 3

Problem 1:

(a) This problem can be expressed as:

$$\underset{f(\vec{x})}{\operatorname{argmin}} E_{Y|\vec{x}}[L(\vec{y},\vec{f})]$$

$$\text{subject to } \sum_{k=1}^{K} f_k = 0$$

$\because$
$$E_{(Y|\vec{x})}[L(\vec{y},\vec{f})] = \sum_{G=g_k} P(G=g_k)\cdot L(\vec{y},\vec{f}\,|\,G=g_k)$$
$$=\sum_{G=g_k} P(G=g_k)\cdot \exp\left(-\frac{1}{K}\left(-\frac{1}{K-1}\cdot f_1+\cdots+1\cdot f_k+\cdots+\left(-\frac{1}{K-1}\right)\cdot f_K\right)\right)$$
$$=\sum_{G=g_k} P(G=g_k)\cdot \exp\left(-\frac{1}{K-1}\cdot f_k\right)\cdot \exp\left(\frac{1}{K(K-1)}\sum_{k=1}^{K} f_k\right)$$

given $\sum_{k=1}^{K} f_k=0$.
$$E_{Y|\vec{x}}[L(\vec{y},\vec{f})] = \sum_{G=g_k} P(G=g_k)\cdot \exp\left(-\frac{1}{K-1} f_k\right)$$

$\therefore$ The Lagrange form can be written as:

$$\sum_{G=g_k} P(G=g_k)\cdot \exp\left(-\frac{1}{K-1} f_k\right) - \lambda \sum_{k=1}^{K} f_k$$

where λ is the Lagrange multiplier.
Taking derivatives with respect to f_k and λ, we have.

$$\begin{cases} -\frac{1}{K-1}\exp\left(-\frac{f_1(x)}{K-1}\right)\cdot P(G=g_1) - \lambda = 0 \\ \cdots\cdots \\ -\frac{1}{K-1}\exp\left(-\frac{f_K(x)}{K-1}\right)\cdot P(G=g_K) - \lambda = 0 \\ f_1(x)+\cdots+f_K(x) = 0. \end{cases}$$

Solving them by

$$f_k^*(x) = (K-1)\left(\log(P(G=g_k)) - \log[-\lambda(K-1)]\right)$$
$$\log[-\lambda(K-1)] = \frac{1}{K}\sum_{G=g_k}\log(P(G=g_k)).$$

$$\therefore\ f_k^*(x) = (K-1)\left[\log[P(G=g_k)] - \frac{1}{K}\sum_{G=g_i}\log[P(G=g_i)]\right]$$

$$f^* = [f_1^*, f_2^*, \cdots, f_K^*].$$

According to $f_k^*(x)$, $P(G=g_k)$ can be written as:

$$P(G=g_k) = \left[\prod_{G=g_i} [P(G=g_i)]\right]^{\frac{1}{K}} \cdot \exp\left(\frac{1}{K-1}\cdot f_k^*\right)$$

$\because \sum P(G=g_k) = 1$

$\therefore$ $$\sum P(G=g_k) = \left[\prod_{G=g_i} P(G=g_i)\right]^{\frac{1}{K}} \cdot \sum_{k=1}^{K} \exp\left(\frac{1}{K-1}\cdot f_k^*\right) = 1$$

$\therefore$ $$P(G=g_k) = \frac{\exp\left(\frac{1}{K-1}\cdot f_k^*\right)}{\sum_{i=1}^{K} \exp\left(\frac{1}{K-1}\cdot f_i^*\right)}$$

$$= \frac{\exp\left(\frac{1}{K-1}\cdot f_k^*\right)}{\sum_{i=1}^{K} \exp\left(\frac{1}{K-1}\cdot f_i^*\right)}$$

(b) From page 343 on "Elements of Statistical Learning", we know that AdaBoost.M1 is equivalent to forward stagewise additive modeling using Loss function

$$L(y, f(x)) = \exp(-y \cdot f(x))$$

Therefore, we can use this method on multi-class exponential loss function to obtain a new algorithm similar to AdaBoost.

Assuming. $\vec{f}(x) = \sum_{m=1}^{M} \beta_m \cdot \vec{b}_m(x)$

where β_m are coefficients, and $\vec{b}_m(x)$ are basis functions. And $\vec{b}(x) \to \vec{y}_i$,

because $\sum_{k=1}^{K} f_k = 0$, $\quad b_1(x) + \cdots + b_K(x) = 0$

Therefore, given the training data, we want to find a solution such that

$$\min_{f(x)} \sum_{i=1}^{N} L(\vec{y}_i, f(x_i))$$

$$\text{subject to } f_1(x) + \cdots + f_K(x) = 0.$$

$\therefore$ Forward stagewise additive modeling will be to solve.

$$(\beta_m, \vec{b}(x)) = \arg\min_{\beta, \vec{b}} \sum_{i=1}^{N} \exp\left(-\frac{1}{K} \vec{y}_i^T (\vec{f}_{m-1}(x_i) + \beta \vec{b}(x_i))\right) \qquad (b\text{-}1)$$

$$= \arg\min_{\beta, \vec{b}} \sum_{i=1}^{N} w_i \cdot \exp\left(-\frac{1}{K} \cdot \beta \cdot \vec{y}_i^T \cdot \vec{b}(x_i)\right)$$

In [1]:
```
import numpy as np
import pandas as pd
from sklearn import tree
from sklearn import model_selection
from sklearn import preprocessing
import matplotlib.pyplot as plt
from sklearn.externals.six import StringIO
import pydot
from IPython.display import Image
from sklearn import metrics
from mpl_toolkits.mplot3d import Axes3D
from sklearn.utils import shuffle
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
%matplotlib inline
```

Problem 2

Decision Tree to Predict Census Income

(a) Pre-processing

In [3]:
```
names = ['age','workclass','fnlwgt','education','education-num','marital
-status','occupation','relationship','race','sex','capital-gain','capita
l-loss','hours-per-week','native-country','label']
adult_data = pd.read_csv('adult.data',header=None, names=names,skipiniti
alspace =True)
test_data = pd.read_csv('adult.test',header=None, names=names,skipinitia
lspace =True)
test_data = test_data.dropna()
test_data['label'] = test_data['label'].str.rsplit('.',expand=True)[0]
adult_data = adult_data.append(test_data,ignore_index=True)
# let's drop education name because they can be defined as education-num
adult_data.drop(adult_data.columns[[3]], axis=1, inplace=True)
adult_data.shape
```

Out[3]: (48842, 14)

In [4]:
```
# fill missing data among the continuous data, though I did not find any
 continuous data is missing.
for item in ['age','fnlwgt','education-num','capital-gain','capital-los
s','hours-per-week']:
    x,y = adult_data.loc[adult_data[item].isnull()].shape
    if x > 0:
        temp_mean = adult_data[item].mean()
        adult_data[item].fillna(temp_mean)
```

Because DecisionTreeClassifier in sklearn can only receive inputs of numbers, the categorical values need to be transfered as dummies values.

In [5]:
```
dummies = {}
for item in ['workclass','marital-status','occupation','relationship','r
ace','sex','native-country','label']:
    miss_index = (adult_data.loc[adult_data[item].str.find('?') != -1].i
ndex.tolist())
    temp_dummy = None
    if len(miss_index) > 0:
        temp_dummy = pd.get_dummies(adult_data[item],
prefix=item).iloc[:, 1:] #drop additional column '?'
        max_mode_index = np.argmax(temp_dummy.sum())
        for mi in miss_index:
            temp_dummy.set_value(mi,max_mode_index,1)
        temp_dummy.drop(temp_dummy.columns[[0]], axis=1, inplace=True)#d
rop one feature which can be represented by all zeros
    else:
        temp_dummy = pd.get_dummies(adult_data[item],
prefix=item).iloc[:, 1:] #drop additional column represented by all zero
s
    dummies[item] = temp_dummy
```

In [6]:
```
# concatenate the dummy variables and drop the duplicates
for key,value in dummies.items():
    if key != 'label':
        adult_data.drop(key, axis=1, inplace=True)
        adult_data = pd.concat([adult_data, value], axis=1)
adult_data.drop('label', axis=1, inplace=True)
adult_data = pd.concat([adult_data, dummies['label']], axis=1)
adult_y = adult_data['label_>50K'].values
adult_data.drop('label_>50K',axis=1, inplace=True)
```

because the size of this train data set is large enough,

I set the rate of train-test as 2:1

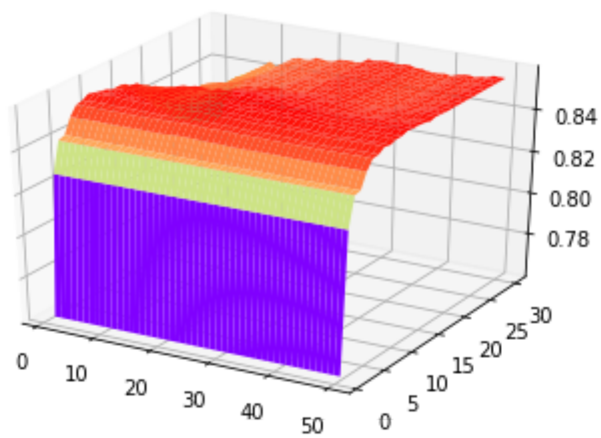
In [7]:
```
trainX, testX, trainY, testY = model_selection.train_test_split(adult_da
ta, adult_y, test_size=1.0/3, random_state=5)
```

(b) Max_depth and Min_samples_leaf

In [11]:
```python
clf = tree.DecisionTreeClassifier()
x_index = range(1,51)
y_index = range(1,31)
X,Y = np.meshgrid(x_index,y_index)
scores = np.zeros(X.shape)
for j in x_index: # min_samples_leaf = j
    for k in y_index: # max_depth = k
        clf.set_params(max_depth=k,min_samples_leaf=j)
        scores[k-1][j-1] = cross_val_score(clf, trainX, trainY,
scoring='accuracy', cv=5).mean()
```

In [12]:
```python
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X,Y,scores,rstride=1, cstride=1, cmap='rainbow')
```

Out[12]:
```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x1144878d0>
```



In [16]:
```python
max_y = np.argmax(np.max(scores,axis=1))+1
max_x = np.argmax(np.max(scores,axis=0))+1
print 'Optimal Choices of parameters:'
print 'max_depth\t','min_samples_leaf'
print max_y,'\t\t\t',max_x
```

```
Optimal Choices of parameters:
max_depth       min_samples_leaf
13                      22
```

(c) Performance on the test set

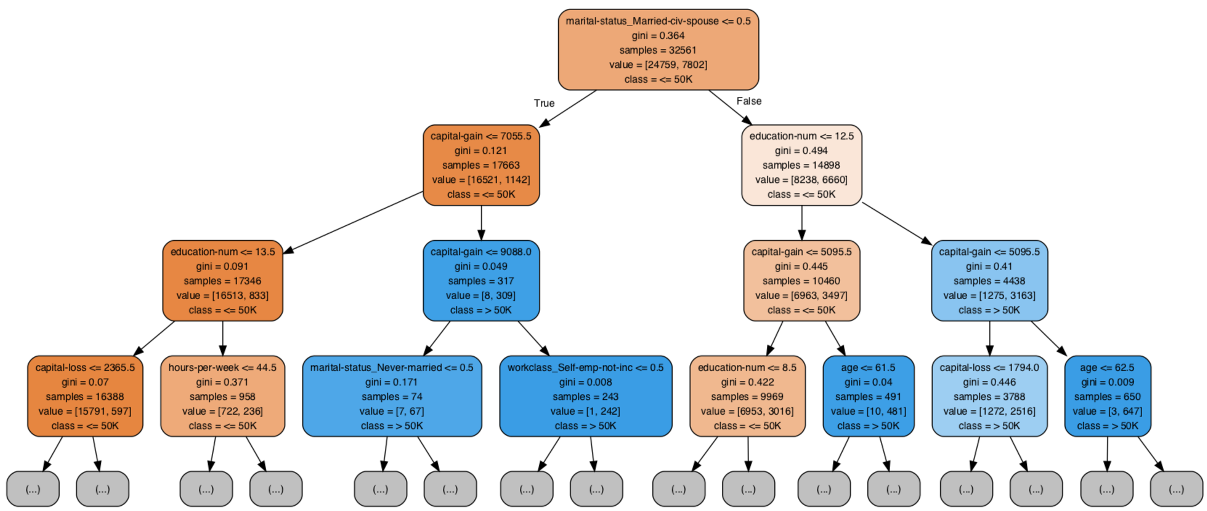
In [17]:
```python
clf.set_params(max_depth=max_y,min_samples_leaf=max_x)
clf.fit(trainX, trainY)
test_score = clf.score(testX,testY)
print 'test_score with optimal parameters'
print test_score
```

```
test_score with optimal parameters
0.857441189116
```

(d) Visualization of the top 3 levels of the D-tree

In [18]:
```
dot_data = StringIO()
tree.export_graphviz(clf, out_file=dot_data,max_depth=3,feature_names=ad
ult_data.columns,class_names=['<= 50K',">
50K"],rounded=True,filled=True)
graph = pydot.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

Out[18]:



Problem 3 Gradient Boosting to Predict Blog Feedback

In [2]:
```
train_data = pd.read_csv('blogData_train.csv', header=None)
validate_data = pd.read_csv('blogData_validate.csv',  header=None)
test_data = pd.read_csv('blogData_test.csv',  header=None)

train_X = train_data.as_matrix(range(280))
validate_X = validate_data.as_matrix(range(280))
test_X = test_data.as_matrix(range(280))

train_Y = np.ravel(train_data.as_matrix([280]))
validate_Y = np.ravel(validate_data.as_matrix([280]))
test_Y = np.ravel(test_data.as_matrix([280]))

scaler = preprocessing.MinMaxScaler()
print(scaler.fit(train_X))

train_X = scaler.transform(train_X)
validate_X = scaler.transform(validate_X)
test_X = scaler.transform(test_X)
```

```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

(a) Implement Gradient Boosting

In [36]:

```
# function of calculate Gradient Boosting
def GBS_mse(x, y, x2, y2, M, v):
    reg = tree.DecisionTreeRegressor(criterion='mse',max_depth = 20,min_
samples_leaf=20)
    f = np.ones(len(y))*np.mean(y) # initial f0
    f2 = np.ones(len(y2))*np.mean(y2)
    err = []
    for m in range(M):
        r = (y-f)
        reg.fit(x,r)
        h = reg.predict(x)
        h2 = reg.predict(x2)
        f = f + v*h
        f2 = f2 + v*h2
        if (m+1) % 5 == 0:
            err.append(metrics.mean_squared_error(y2,f2))
    return err

def GBS_mae(x, y, x2, y2, M, v):
    reg = tree.DecisionTreeRegressor(criterion='mse',max_depth = 20,min_
samples_leaf=20)
    f = np.ones(len(y))*np.mean(y) # initial f0
    f2 = np.ones(len(y2))*np.mean(y2)
    err = []
    for m in range(M):
        r = np.sign((y-f))
        reg.fit(x,r)
        h = reg.predict(x)
        h2 = reg.predict(x2)
        f = f + v*h
        f2 = f2 + v*h2
        if (m+1) % 5 == 0:
            err.append(metrics.mean_absolute_error(y2,f2))
    return err
```

In [8]:
```
m = 25
v = 0.1

f_mse = GBS_mse(train_X, train_Y, validate_X, validate_Y, m,v)
f_mae = GBS_mae(train_X, train_Y, validate_X, validate_Y, m,v)

print 'on validate set (v = 0.1)'
table_a = pd.DataFrame(data=[f_mse,f_mae], index=['MSE','MAE'], columns=
['nIter=5','nIter=10','nIter=15','nIter=20','nIter=25'])
table_a.head()
```

on validate set (v = 0.1)

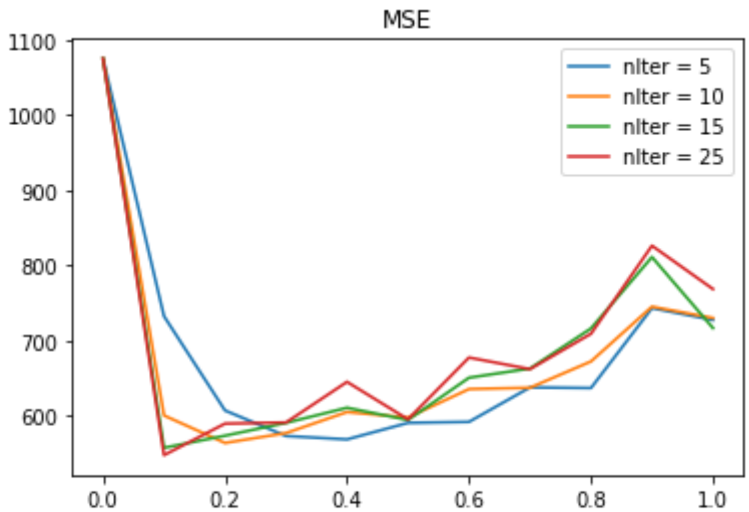
Out[8]:

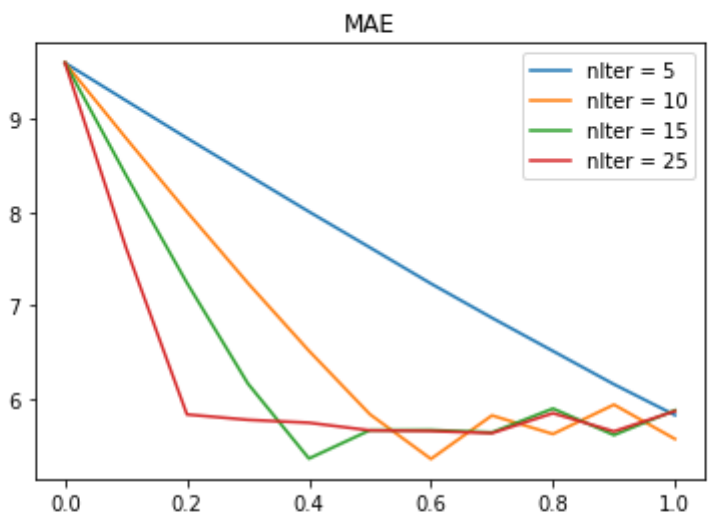
	nIter=5	nIter=10	nIter=15	nIter=20	nIter=25
MSE	753.452293	632.170568	580.031906	561.109836	550.498608
MAE	9.200012	8.794214	8.397229	8.004901	7.619146

(b) Number of boosting iteration and Shrinkage parameter

In [37]:
```
M = 25
V = np.linspace(0,1,11)

mse_arr = []
mae_arr = []
for v in V:
    temp_mse_err = GBS_mse(train_X, train_Y, validate_X, validate_Y,
M,v)
    temp_mae_err = GBS_mae(train_X, train_Y, validate_X, validate_Y,
M,v)
    mse_arr.append(temp_mse_err)
    mae_arr.append(temp_mae_err)

mse_arr = np.array(mse_arr)
mae_arr = np.array(mae_arr)
# delete column nIter = 20
mse_arr = np.delete(mse_arr,3,1)
mae_arr = np.delete(mae_arr,3,1)
```

In [38]:
```
plt.figure(1)

plt.plot(V,np.ravel(mse_arr[:,0]),label='nIter = 5')
plt.plot(V,np.ravel(mse_arr[:,1]),label='nIter = 10')
plt.plot(V,np.ravel(mse_arr[:,2]),label='nIter = 15')
plt.plot(V,np.ravel(mse_arr[:,3]),label='nIter = 25')
plt.title('MSE')
plt.legend()
plt.show()

plt.plot(V,np.ravel(mae_arr[:,0]),label='nIter = 5')
plt.plot(V,np.ravel(mae_arr[:,1]),label='nIter = 10')
plt.plot(V,np.ravel(mae_arr[:,2]),label='nIter = 15')
plt.plot(V,np.ravel(mae_arr[:,3]),label='nIter = 25')
plt.title('MAE')
plt.legend()
plt.show()
```





Conlusion

(1) Comparing the validation errors for MSE and MAE, we can see that the convergence speed of MSE is faster than that of MAE such that the optimal shrinkage of MSE is smaller than that of MAE. I think this is because the step of each gradient in MSE depends on (y-f), while the step of each gradient in MAE has been constrained by one pace (without shrinkage) such that the learning speed of MAE is smaller than that of MSE.

(2) Looking at the parameter nIter, there are some difference between MSE and MAE.

For MSE, given fixed small shrinkage v, we can see that the models with small nIter have higher validation error than the models with large nIter and vise versa. I think this is because smaller v have better shrinkage performance.

For MAE, when the nIter is small, the model cannot achieve good performance because its low learning speed(see the curve as nIter = 5). But when nIter is too large, the model cannot achieve good performance maybe because it overfit in train set. So the optimal value of nIter in MAE is 10 here.

(3) Looking at the shrinkage v, given fixed nIter, we can see that the validation errors decrease as the shrinkage v increase from 0, but it begins to increase when the shrinkage v reach to a certain value. It is obvious that if v = 0, there is no gradient boosting such that validation errors are high. And if v = 1, there is no shrinkage on gradient boosting such that the influence of a new classifier is too large such that the model cannot obtain a better performance. However, the particular case of MAE with nIter = 5 does not show this property. I think this is because the model with MAE Gradient boosting does not achieve optimal result after 5 iteration of Gradient boosting.

(4) According to the analysis in (2) and (3), we can conclude that the nIter and the shrinkage are complementary which means the smaller shrinkage v requires the larger nIter to obtain the better performance. Therefore, to achieve a tradeoff to obtain a better performance, we can use a larger shrinkage with smaller nIter, or use a smaller shrinkage with larger nIter. The chioce of these two cases depends on the loss function and the data.

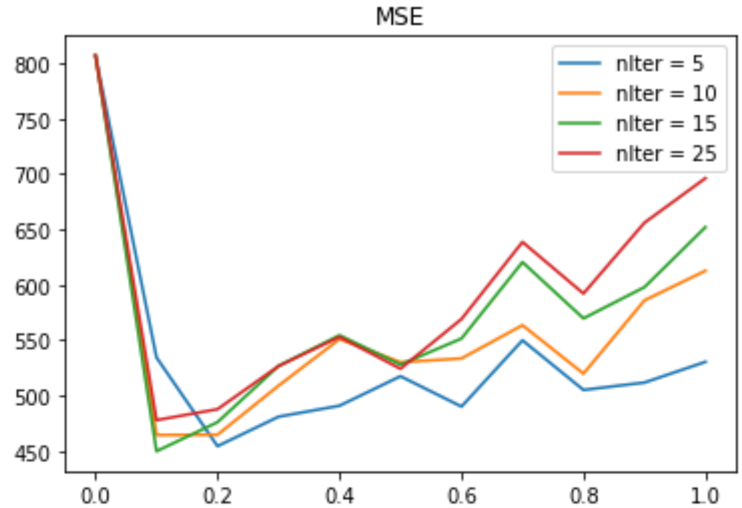
(c)

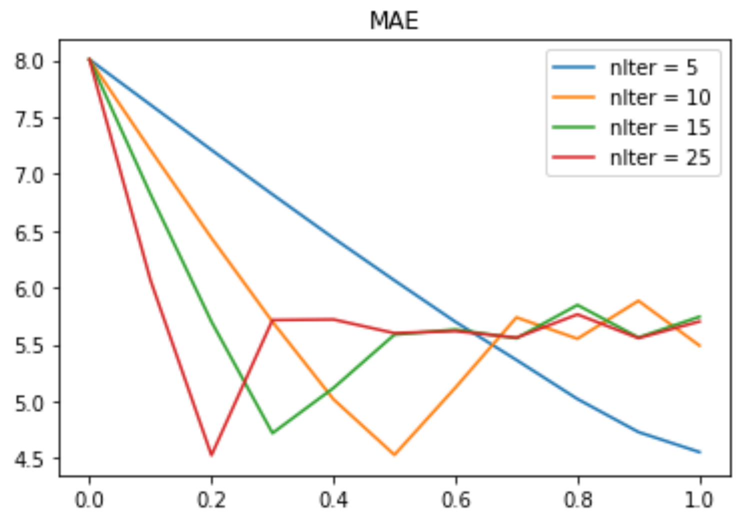
In [39]:
```
nIters = [5,10,15,25]
m_min_mse = nIters[np.argmin(mse_arr)%4]
v_min_mse = V[np.argmin(mse_arr)/5]

m_min_mae = nIters[np.argmin(mae_arr)%4]
v_min_mae = V[np.argmin(mae_arr)/5]
```

In [40]:
```
M = 25
V = np.linspace(0,1,11)

test_mse_arr = []
test_mae_arr = []
for v in V:
    temp_mse_err = GBS_mse(train_X, train_Y, test_X, test_Y, M,v)
    temp_mae_err = GBS_mae(train_X, train_Y, test_X, test_Y, M,v)
    test_mse_arr.append(temp_mse_err)
    test_mae_arr.append(temp_mae_err)

test_mse_arr = np.array(test_mse_arr)
test_mae_arr = np.array(test_mae_arr)
# delete column nIter = 20
test_mse_arr = np.delete(test_mse_arr,3,1)
test_mae_arr = np.delete(test_mae_arr,3,1)
```

In [41]:
```
f_mse = GBS_mse(train_X, train_Y, test_X, test_Y, m_min_mse,v_min_mse)
[-1]
f_mae = GBS_mae(train_X, train_Y, test_X, test_Y, m_min_mae,v_min_mae)
[-1]
```

In [42]:
```
plt.figure(1)

plt.plot(V,np.ravel(test_mse_arr[:,0]),label='nIter = 5')
plt.plot(V,np.ravel(test_mse_arr[:,1]),label='nIter = 10')
plt.plot(V,np.ravel(test_mse_arr[:,2]),label='nIter = 15')
plt.plot(V,np.ravel(test_mse_arr[:,3]),label='nIter = 25')
plt.title('MSE')
plt.legend()
plt.show()

plt.plot(V,np.ravel(test_mae_arr[:,0]),label='nIter = 5')
plt.plot(V,np.ravel(test_mae_arr[:,1]),label='nIter = 10')
plt.plot(V,np.ravel(test_mae_arr[:,2]),label='nIter = 15')
plt.plot(V,np.ravel(test_mae_arr[:,3]),label='nIter = 25')
plt.title('MAE')
plt.legend()
plt.show()


print 'on test set '
print 'MSE(v_max_valid = ',v_min_mse, ',nIter_max_valid = ',m_min_mse,')
 : \t', f_mse
print 'MAE(v_max_valid = ',v_min_mae, ',nIter_max_valid = ',m_min_mae,')
 : \t', f_mae
```






```
on test set
MSE(v_max_valid =  0.1 ,nIter_max_valid =  25 ) :       476.066011849
MAE(v_max_valid =  0.5 ,nIter_max_valid =  10 ) :       4.52752796832
```

Conclusion

As you can see in the above graph, although the model with optimal parameter in validation set cannont achieve best performance in test set because the two data set are different and the model is not too robust, it still achieve good enough performance in test set. Moreover, at the same nIter, the shrinkage v of the lowest rate in test set is the same as the shrinkage v of the lowest rate in validation set.

where $w_i = \exp\left(-\frac{1}{K}\vec{y}_i^T\cdot\vec{f}_{m-1}(x_i)\right)$ are weights.

Because every $\vec{b}(x)$ has a one-to-one correspondence with a multi-class classifier $G(x)$, and $\vec{b}(x_i) \to \vec{Y}$, $\vec{y} \in \vec{Y}$, $y_K = \begin{cases} 1 & \text{if } G = g_k \\ -\frac{1}{K-1} & \text{otherwise.} \end{cases}$

∴ Set $G(x) = g_k$, if $b_k(x) = 1$. i.e. $b_k(x) = \begin{cases} 1 & \text{if } G(x) = g_k \\ -\frac{1}{K-1} & \text{if } G(x) \neq g_k. \end{cases}$

So $$G_m(x) = \arg\min_{G} \sum_{i=1}^{N} w_i^{(m)} \vec{I}(G(x_i) \neq g_k) \qquad \text{(b-2)}$$

which is the classifier that minimizes the weighted error rate in predicting $\vec{y}$.

It can be conducted from (b-1):

$$\sum_{G(x_i)=g_k} w_i \cdot e^{-\frac{\beta}{K-1}} + \sum_{G(x_i)\neq g_k} w_i \cdot e^{\frac{\beta}{(K-1)^2}} \qquad \text{(b-3)}$$

$$= e^{-\frac{\beta}{K-1}} \sum_i w_i + \left(e^{\frac{\beta}{(K-1)^2}} - e^{-\frac{\beta}{K-1}}\right) \sum_i w_i \cdot \vec{I}(G(x_i) \neq g_k).$$

$$\vec{y}_i^T \cdot \vec{b}(x_i) = \begin{cases} \frac{K}{K-1}, & G(x_i) = g_k \\ \frac{-K}{(K-1)^2}, & G(x_i) \neq g_k. \end{cases}$$

Plugging $G_m(x)$ in (b-2) into (b-1), and solving for β.

$$\beta_m = \frac{(K-1)^2}{K}\left(\log\frac{1-err_m}{err_m} + \log(K-1)\right) \qquad \text{(b-4)}$$

where err_m is defined as.

$$err_m = \sum_{i=1}^{N} w_i \vec{I}(G(x_i) \neq g_k) \Big/ \sum_{i=1}^{N} w_i$$

(b-4) can be obtained by making gradient to (b-3) of β 0.

$$-\frac{1}{K-1} e^{-\frac{\beta}{K-1}} \sum w_i + \left(\frac{1}{(K-1)^2} \cdot e^{\frac{\beta}{(K-1)^2}} + \frac{1}{K-1} e^{-\frac{\beta}{K-1}}\right) \sum w_i \cdot \vec{I}(G(x_i) \neq g_k) = 0$$

let $err = \frac{\sum w_i \cdot \vec{I}(G(x_i) \neq g_k)}{\sum w_i}$, we have.

$$-e^{-\frac{\beta}{K-1}} + \left(\frac{1}{K-1} \cdot e^{\frac{\beta}{(K-1)^2}} + e^{-\frac{\beta}{K-1}}\right) err = 0.$$

$$\frac{err}{K-1} \cdot e^{\frac{\beta}{(K-1)^2}} = (1-err) e^{-\frac{\beta}{K-1}}$$

$$\log err - \log(K-1) + \frac{\beta}{(K-1)^2} = \log(1-err) + \left(-\frac{\beta}{K-1}\right)$$

$$\frac{K}{(K-1)^2}\beta = \log\left(\frac{1-err}{err}\right) + \log(K-1)$$

$$\beta = \frac{(K-1)^2}{K}\left(\log\left(\frac{1-err}{err}\right) + \log(K-1)\right)$$

Then, the model will be updated

$$\vec{f}_m(x) = \vec{f}_{m-1}(x) + \beta_m \cdot \vec{b}(x)$$

and the weights will be updated

$$w_i^{(m+1)} = w_i^{(m)} \cdot \exp\left(-\frac{1}{K}\beta_m \cdot \vec{y}_i^T \cdot \vec{b}(x_i)\right).$$

according to β_m in (b-4).

$$w_i^{(m+1)} = w_i^{(m)} \cdot \exp\left(-\frac{(K-1)^2}{K^2} \cdot \left(\log\frac{1-err}{err} + \log(K-1)\right) \cdot \vec{y}_i^T \cdot \vec{b}(x_i)\right)$$

Because $\vec{y}_i^T \cdot \vec{b}(x_i) = \begin{cases} \frac{K}{K-1}, & G(x_i) = g_k \\ \frac{-K}{(K-1)^2}, & G(x_i) \neq g_k \end{cases}$

$$\therefore \quad w_i^{(m+1)} = \begin{cases} w_i^{(m)} \cdot \exp\left(-\frac{K-1}{K} \cdot \left(\log\frac{1-err}{err} + \log(K-1)\right)\right), & G(x_i) = g_k \\ w_i^{(m)} \cdot \exp\left(\frac{1}{K}\left(\log\frac{1-err}{err} + \log(K-1)\right)\right), & G(x_i) \neq g_k \end{cases}$$

Therefore, compare to AdaBoost's. $\alpha_m = \log((1-err_m)/err_m)$.

this new $\alpha^{(m)}$ can be

$$\alpha^{(m)}_{new} = \log\left(\frac{1-err}{err}\right) + \log(K-1) = \alpha^{(m)}_{old} + \log(K-1)$$