Machine Learning Lab 04: Multinomial Logistic Regression

Generalized Linear Models

From lecture, we know that members of the exponential family distributions can be written in the form

$$p(y;\eta) = b(y)e^{(\eta^\top T(y) - a(\eta))},$$

where

- η is the natural parameter or canonical paramter of the distribution,
- $T(y)$ is the sufficient statistic (we normally use $T(y) = y$),
- $b(y)$ is an arbitrary scalar function of y, and
- $a(\eta)$ is the log partition function. We use $e^{a(\eta)}$ just to normalize the distribution to have a sum or integral of 1.

Each choice of T, a, and b defines a family (set) of distributions parameterized by η.

If we can write $p(y \mid \mathbf{x}; \theta)$ as a member of the exponential family of distributions with parameters η with $\eta_i = \theta_i^\top \mathbf{x}$, we obtain a *generalized linear model* that can be optimized using the maximum likelihood principle.

The GLM for the Gaussian distribution with natural parameter η being the mean of the Gaussian gives us ordinary linear regression.

The Bernoulli distribution with parameter ϕ can be written as an exponential distribution with natural parmeter $\eta = \log \frac{\phi}{1-\phi}$. The GLM for this distribution is logistic regression.

When we write the multinomial distribution with paremeters $\phi_i > 0$ for classes $i \in 1..K$ with the constraint that

$$\sum_{i=1}^{K} \phi_i = 1$$

as a member of the exponential family, the resulting GLM is called *multinomial logistic regression*. The parameters $\phi_1, \ldots, \phi_K$ are written in terms of θ as

$$\phi_i = p(y = i \mid \mathbf{x}; \theta) = \frac{e^{\theta_i^\top \mathbf{x}}}{\sum_{j=1}^{K} e^{\theta_j^\top \mathbf{x}}}.$$

Optimizing a Multinomial Regression Model

In multinomial regression, we have

1. Data are pairs $\mathbf{x}^{(i)}, y^{(i)}$ with $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and $y \in 1..K$.
2. The hypothesis is a vector-valued function $$\mathbf{h}_\theta(\mathbf{x}) = \begin{bmatrix} p(y = 1 \mid \mathbf{x} ; \theta) \\ p(y = 2 \mid \mathbf{x} ; \theta) \\ \vdots \\ p(y = K \mid \mathbf{x} ; \theta) \end{bmatrix},$$

   where

$$p(y = i \mid \mathbf{x}) = \phi_i = p(y = i \mid \mathbf{x}; \theta) = \frac{e^{\theta_i^\top \mathbf{x}}}{\sum_{j=1}^K e^{\theta_j^\top \mathbf{x}}}.$$

We need a cost function and a way to minimize that cost function. As usual, we try to find the parameters maximizing the likelihood or log likelihood function, or equivalently, minimizing the negative log likelihood function:

$$\theta^* = \mathrm{argmax}_\theta \mathcal{L}(\theta) = \mathrm{argmax}_\theta \ell(\theta) = \mathrm{argmin}_\theta J(\theta),$$

where

$$\begin{aligned} J(\theta) &= -\ell(\theta) \\ &= -\sum_{i=1}^m \log p(y^{(i)} \mid \mathbf{x}^{(i)}; \theta). \end{aligned}$$

Now that we know what is $J(\theta)$, let's try to find its minimimum by taking the derivatives with respect to an arbitrary parameter θ_{kl}, the l-th element of the parameter vector θ_k for class k. Before we start, let's define a variable a_k as the linear activation for class k in the softmax function:

$$a_k = \theta_k^\top \mathbf{x}^{(i)},$$

and rewrite the softmax more conveniently as

$$\phi_k = \frac{e^{a_k}}{\sum_{j=1}^K e^{a_j}}.$$

That makes it a little easier to compute the gradient:

$$\frac{\partial J}{\partial \theta_{kl}} = -\sum_{i=1}^m \frac{1}{\phi_{y^{(i)}}} \frac{\partial \phi_{y^{(i)}}}{\partial \theta_{kl}}.$$

Using the chain rule, we have

$$\frac{\partial \phi_{y^{(i)}}}{\partial \theta_{kl}} = \sum_{j=1}^K \frac{\partial \phi_{y^{(i)}}}{\partial a_j} \frac{\partial a_j}{\partial \theta_{kl}}$$

The second factor is easy:

$$\frac{\partial a_j}{\partial \theta_{kl}} = \delta(k = j) x_l^{(i)}.$$

For the first factor, we have

$$\begin{aligned}\frac{\partial \phi_{y^{(i)}}}{\partial a_j} &= \frac{\left[\delta(y^{(i)} = j) e^{a_j} \sum_{c=1}^{K} e^{a_c}\right] - e^{a_j} e^{a_j}}{\left[\sum_{c=1}^{K} e^{a_c}\right]^2} \\ &= \delta(y^{(i)} = j)\phi_j - \phi_j^2\end{aligned}$$

Substituting what we've derived into the definition above, we obtain

$$\frac{\partial J}{\theta_{kl}} = -\sum_{i=1}^{m}\sum_{j=1}^{K}(\delta(y^{(i)} = j) - \phi_j)\frac{\partial a_j}{\partial \theta_{kl}}.$$

There are two ways to do the calculation. In deep neural networks with multinomial outputs, we want to first calculate the $\frac{\partial J}{\partial a_j}$ terms then use them to calculate $\frac{\partial J}{\partial \theta_{kl}}$.

However, if we only have the "single layer" model described up till now, we note that

$$\frac{\partial a_j}{\partial \theta_{kl}} = \delta(j = k)x_l^{(i)},$$

so we can simplify as follows:

$$\begin{aligned}\frac{\partial J}{\theta_{kl}} &= -\sum_{i=1}^{m}\sum_{j=1}^{K}(\delta(y^{(i)} = j) - \phi_j)\frac{\partial a_j}{\partial \theta_{kl}} \\ &= -\sum_{i=1}^{m}\sum_{j=1}^{K}(\delta(y^{(i)} = j) - \phi_j)\delta(j = k)x_l^{(i)} \\ &= \sum_{i=1}^{m}(\delta(y^{(i)} = k) - \phi_k)x_l^{(i)}\end{aligned}$$

Put It Together

OK! Now we have all 4 criteria for our multinomial regression model:

1. Data are pairs $\mathbf{x}^{(i)}, y^{(i)}$ with $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and $y \in 1..K$.
2. The hypothesis is a vector-valued function $$\mathbf{h}_\theta(\mathbf{x}) = \begin{bmatrix} p(y = 1 \mid \mathbf{x} ; \theta) \\ p(y = 2 \mid \mathbf{x} ; \theta) \\ \vdots \\ p(y = K \mid \mathbf{x} ; \theta) \end{bmatrix},$$

   where

$$p(y = i \mid \mathbf{x}) = \phi_i = p(y = i \mid \mathbf{x}; \theta) = \frac{e^{\theta_i^\top \mathbf{x}}}{\sum_{j=1}^K e^{\theta_j^\top \mathbf{x}}}.$$

3. The cost function is

$$J(\theta) = -\sum_{i=1}^m \log p(y^{(i)} \mid \mathbf{x}^{(i)})$$

4. The optimization algorithm is gradient descent on $J(\theta)$ with the update rule

$$\theta_{kl}^{(n+1)} \leftarrow \theta_{kl}^{(n)} - \alpha \sum_{i=1}^m (\delta(y^{(i)} = k) - \phi_k) x_l^{(i)}.$$

Multinomial Regression Example

The following example of multinomial logistic regression is from Kaggle (https://www.kaggle.com/saksham219/softmax-regression-for-iris-classification).

The data set is the famous Iris dataset from the UCI machine learning repository (https://archive.ics.uci.edu/ml/datasets/iris).

The data contain 50 samples from each of three classes. Each class refers to a particular species of the iris plant. The data include four independent variables:

1. Sepal length in cm
2. Sepal width in cm
3. Petal length in cm
4. Petal width in cm

The target takes on one of three classes:

1. Iris Setosa
2. Iris Versicolour
3. Iris Virginica

To predict the target value, we use multinomial logistic regression for $k = 3$ classes i.e. $y \in \{1, 2, 3\}$.

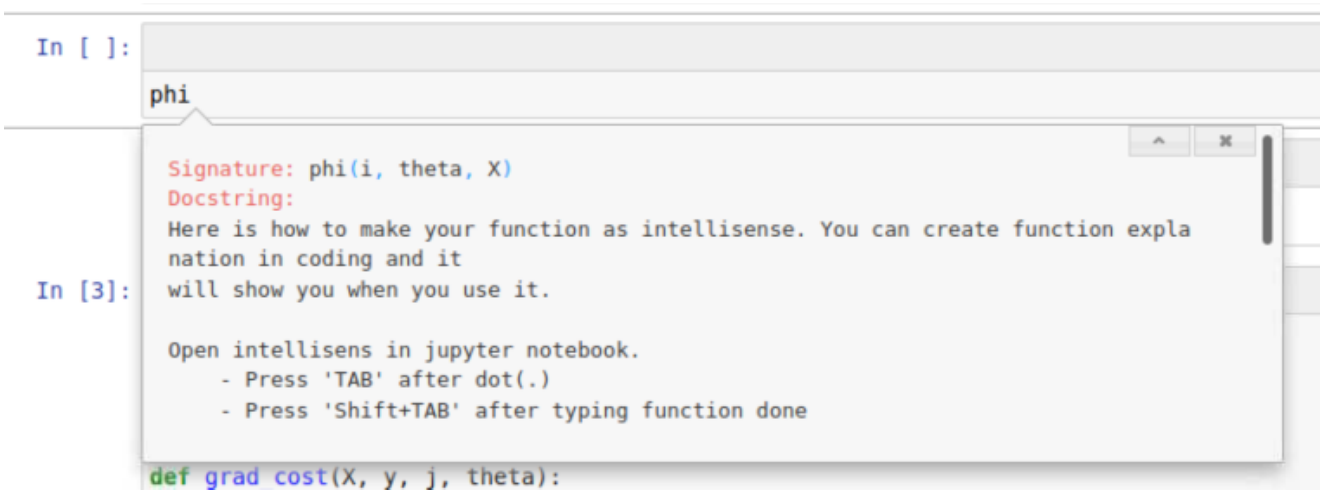
Given $\mathbf{x}$, we would like to predict a probability distribution over the three outcomes for y, i.e., $\phi_1 = p(y = 1 \mid \mathbf{x})$, $\phi_2 = p(y = 2 \mid \mathbf{x})$, and $\phi_3 = p(y = 3 \mid \mathbf{x})$.

In [7]:
```
# importing libraries
import numpy as np
import pandas as pd
import random
import math
```

The `phi` function returns ϕ_i for input patterns $\mathbf{X}$ and parameters θ.

In [8]:
```
def phi(i, theta, X, num_class):
    """
    Here is how to make documentation for your function show up i
n intellisense.
    Explanation you put here will be shown when you use it.

    To get intellisense in your Jupyter notebook:
        - Press 'TAB' after typing a dot (.) to see methods and a
ttributes
        - Press 'Shift+TAB' after typing a function name to see i
ts documentation

    The `phi` function returns phi_i = h_theta(x) for input patte
rns X and parameters theta.

    Inputs:
        i=index of phi

        X=input dataset

        theta=parameters

    Returns:
        phi_i
    """
    mat_theta = np.matrix(theta[i])
    mat_x = np.matrix(X)
    num = math.exp(np.dot(mat_theta, mat_x.T))
    den = 0
    for j in range(0,num_class):
        mat_theta_j = np.matrix(theta[j])
        den = den + math.exp(np.dot(mat_theta_j, mat_x.T))
    phi_i = num / den
    return phi_i
```

Tips for using intellisense: Shift+TAB



The `grad_cost` function gives the gradient of the cost for data $\mathbf{X}, \mathbf{y}$ for class $j \in 1..k$.

In [23]:
```
def indicator(i, j):
    '''
    Check whether i is equal to j

    Return:
        1 when i=j, otherwise 0
    '''
    if i == j: return 1
    else: return 0


def grad_cost(X, y, j, theta, num_class):
    '''
    Compute the gradient of the cost function for data X, y for p
arameters of
    output for class j in 1..k
    '''
    m, n = X.shape
    sum = np.array([0 for i in range(0,n)])
    for i in range(0, m):
        p = indicator(y[i], j) - phi(j, theta, X.loc[i], num_clas
s)
        sum = sum + (X.loc[i] * p)
    grad = -sum / m
    return grad

def gradient_descent(X, y, theta, alpha, iters, num_class):
    '''
    Perform iters iterations of gradient descent: theta_new = the
ta_old - alpha * cost
    '''
    n = X.shape[1]
    for iter in range(iters):
        dtheta = np.zeros((num_class, n))
        for j in range(0, num_class):
            dtheta[j,:] = grad_cost(X, y, j, theta, num_class)
        theta = theta - alpha * dtheta
    return theta

def h(X, theta, num_class):
    '''
    Hypothesis function: h_theta(X) = theta * X
    '''
    X = np.matrix(X)
    h_matrix = np.empty((num_class,1))
    den = 0
    for j in range(0, num_class):
        den = den + math.exp(np.dot(theta[j], X.T))
    for i in range(0,num_class):
        h_matrix[i] = math.exp(np.dot(theta[i], X.T))
    h_matrix = h_matrix / den
    return h_matrix
```

Exercise 1.1 (5 points)

Create a function to load **data** from **Iris.csv** using the Pandas library and extract y from the data.

You can use the Pandas 10 minute guide (https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html) to learn how to use pandas.

In [16]:
```
def load_data(file_name, drop_label, y_label, is_print=False):
    # 1. Load csv file
    data = pd.read_csv(file_name)
    if is_print:
        print(data.head())
    # 2. remove 'Id' column from data
    if drop_label is not None:
        data = data.drop([drop_label],axis=1)
        if is_print:
            print(data.head())
    # 3. Extract y_label column as y from data
    y = None
    # 4. get index of y-column
    y_index = data.columns.get_loc(y_label)
    # 5. Extrack X features from data
    X = None
    ### BEGIN SOLUTION
    y = data[y_label]
    X = data.iloc[:,:y_index]
    ### END SOLUTION
    return X, y
```

In [17]:
```
X, y = load_data('Iris.csv', 'Id', 'Species', True)
print(X.head())
print(y[:5])

# Test function: Do not remove
# tips: this is how to create dataset using pandas
d_ex = {'ID':      [  1,   2,   3,    4,    5,    6,    7],
        'Grade':  [3.5, 2.5, 3.0, 3.75, 2.83, 3.95, 2.68],
        'Type':   ['A', 'B', 'C',  'A',  'C',  'A',  'B']
        }
df = pd.DataFrame (d_ex, columns = ['ID','Grade', 'Type'])
df.to_csv('out.csv', index=False)

Xtest, ytest = load_data('out.csv', 'ID', 'Type')
assert len(Xtest.columns) == 1, 'number of X_columns incorrect
(1)'
assert ytest.name == 'Type', 'Extract y_column is incorrect (1)'
assert ytest.shape == (7,), 'number of y is incorrect (1)'
assert 'Grade' in Xtest.columns, 'Incorrect columns in X (1)'
Xtest, ytest = load_data('out.csv', None, 'Type')
assert len(Xtest.columns) == 2, 'number of X_columns incorrect
(2)'
assert ytest.name == 'Type', 'Extract y_column is incorrect (2)'
assert ytest.shape == (7,), 'number of y is incorrect (2)'
assert 'Grade' in Xtest.columns and 'ID' in Xtest.columns, 'Incor
rect columns in X (2)'
import os
os.remove('out.csv')

assert len(X.columns) == 4, 'number of X_columns incorrect (3)'
assert 'SepalWidthCm' in X.columns and 'Id' not in X.columns and
'Species' not in X.columns, 'Incorrect columns in X (3)'
assert y.name == 'Species', 'Extract y_column is incorrect (3)'
assert y.shape == (150,), 'number of y is incorrect (3)'

print("success!")
# End Test function
```

```
   Id  SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
Species
0   1            5.1           3.5            1.4           0.2
Iris-setosa
1   2            4.9           3.0            1.4           0.2
Iris-setosa
2   3            4.7           3.2            1.3           0.2
Iris-setosa
3   4            4.6           3.1            1.5           0.2
Iris-setosa
4   5            5.0           3.6            1.4           0.2
Iris-setosa
   SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
Species
0            5.1           3.5            1.4           0.2  Iris
-setosa
1            4.9           3.0            1.4           0.2  Iris
-setosa
2            4.7           3.2            1.3           0.2  Iris
-setosa
3            4.6           3.1            1.5           0.2  Iris
-setosa
4            5.0           3.6            1.4           0.2  Iris
-setosa
   SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
0            5.1           3.5            1.4           0.2
1            4.9           3.0            1.4           0.2
2            4.7           3.2            1.3           0.2
3            4.6           3.1            1.5           0.2
4            5.0           3.6            1.4           0.2
0    Iris-setosa
1    Iris-setosa
2    Iris-setosa
3    Iris-setosa
4    Iris-setosa
Name: Species, dtype: object
success!
```

Expected result: \ SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm \ 0 5.1 3.5 1.4 0.2\ 1 4.9 3.0 1.4 0.2\ 2 4.7 3.2 1.3 0.2\ 3 4.6 3.1 1.5 0.2\ 4 5.0 3.6 1.4 0.2\ 0 Iris-setosa\ 1 Iris-setosa\ 2 Iris-setosa\ 3 Iris-setosa\ 4 Iris-setosa\ Name: Species, dtype: object

Exercise 1.2 (10 points)

Partition data into training and test sets

- No need to use random.seed function!
- Ensure that the train set is 70% and the test set is 30% of the data.
- Encode the labels in the y attribute to be integers in the range 0..k-1.

Hint:

In [18]:
```
def partition(X, y, percent_train):
    # 1. create index list
    # 2. shuffle index
    # 3. Create train/test index
    # 4. Separate X_Train, y_train, X_test, y_test
    # 5. Get y_labels_name from y using pandas.unique function
    # 6. Change y_labels_name into string number and put into y_l
abels_new
    # 7. Drop shuffle index columns
    #     - pandas.reset_index() and pandas.drop(...) might be he
lp

    y_labels_name = None
    y_labels_new = None

    ### BEGIN SOLUTION
    idx = np.arange(0,y.shape[0])
    random.shuffle(idx)

    m_train = int(y.shape[0] * percent_train)
    train_idx = idx[0:m_train]
    test_idx = idx[m_train:y.shape[0]+1]

    X_train = X.iloc[train_idx,:]
    X_test = X.iloc[test_idx,:]

    y_train = y.iloc[train_idx]
    y_test = y.iloc[test_idx]

    y_labels_name = y.unique()

    i = 0
    y_labels_new = []
    for label in y_labels_name:
        y_train[y_train.str.match(label)] = str(i)
        y_test[y_test.str.match(label)] = str(i)
        y_labels_new.append(i)
        i = i + 1
    y_train = y_train.astype(int)
    y_test = y_test.astype(int)

    X_train = X_train.reset_index()
    X_train = X_train.drop(['index'],axis=1)
    X_test = X_test.reset_index()
    X_test = X_test.drop(['index'],axis=1)
    y_train = y_train.reset_index()
    y_train = y_train.drop(['index'],axis=1)
    y_test = y_test.reset_index()
    y_test = y_test.drop(['index'],axis=1)

    y_label = y.name
    y_train = y_train[y_label]
    y_test = y_test[y_label]
    ### END SOLUTION

    return idx, X_train, y_train, X_test, y_test, y_labels_name,
y_labels_new
```

In [19]:
```
percent_train = 0.7
idx, X_train, y_train, X_test, y_test, y_labels_name, y_labels_ne
w = partition(X, y, percent_train)
print('X_train.shape', X_train.shape)
print('X_test.shape', X_test.shape)
print('y_train.shape', y_train.shape)
print('y_test.shape', y_test.shape)
print('y_labels_name: ', y_labels_name)
print('y_labels_new: ', y_labels_new)
print(X_train.head())
print(y_train.head())

# Test function: Do not remove
assert len(y_labels_name) == 3 and len(y_labels_new) == 3, 'numbe
r of y uniques are incorrect'
assert X_train.shape == (105, 4), 'Size of X_train is incorrect'
assert X_test.shape == (45, 4), 'Size of x_test is incorrect'
assert y_train.shape == (105, ), 'Size of y_train is incorrect'
assert y_test.shape == (45, ), 'Size of y_test is incorrect'
assert 'Iris-setosa' in y_labels_name and 'Iris-virginica' in y_l
abels_name and \
        'Iris-versicolor' in y_labels_name, 'y unique data incorr
ect'
assert min(y_labels_new) == 0 and max(y_labels_new) < 3, 'label i
ndices are incorrect'

print("success!")
# End Test function
```

```
X_train.shape (105, 4)
X_test.shape (45, 4)
y_train.shape (105,)
y_test.shape (45,)
y_labels_name:  ['Iris-setosa' 'Iris-versicolor' 'Iris-virginica
']
y_labels_new:  [0, 1, 2]
   SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
0            5.7           2.9            4.2           1.3
1            5.4           3.0            4.5           1.5
2            5.1           3.7            1.5           0.4
3            4.8           3.1            1.6           0.2
4            7.2           3.0            5.8           1.6
0    1
1    1
2    0
3    0
4    2
Name: Species, dtype: int64
success!
```

Expected result: (*or similar*)\ X_train.shape (105, 4)\ X_test.shape (45, 4)\ y_train.shape (105,)\ y_test.shape (45,)\ y_labels_name: ['Iris-setosa' 'Iris-versicolor' 'Iris-virginica'] \ y_labels_new: [0, 1, 2]

SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm\ 0 6.4 2.8 5.6 2.2\ 1 6.7 3.3 5.7 2.1\ 2 4.6 3.4 1.4 0.3\ 3 5.1 3.8 1.5 0.3\ 4 5.0 2.3 3.3 1.0\ Species\ 0 2\ 1 2\ 2 0\ 3 0\ 4 1

Exercise 1.3 (5 points)

Train your classification model using the `gradient_descent` function already provided. You might also play around with the gradient descent function to see if you can speed it up!

In [22]:
```
# num_class is the number of unique labels
num_class = len(y_labels_name)

if (X_train.shape[1] == X.shape[1]):
    X_train.insert(0, "intercept", 1)

# Reset m and n for training data
r, c = X_train.shape

# Initialize theta for each class
theta_initial = np.ones((num_class, c))

alpha = .05
iterations = 200

theta = None
# Logistic regression
### BEGIN SOLUTION
theta = gradient_descent(X_train, y_train, theta_initial, alpha,
iterations, num_class)
### END SOLUTION
```

```
Theta shape: (3, 5)
```

In [21]:
```
print(theta)
print(theta.shape)

# Test function: Do not remove
assert theta.shape == (3, 5), 'Size of theta is incorrect'

print("success!")
# End Test function
```

```
[[ 1.17632192  1.32360047  1.83204165 -0.20224445  0.44039155]
 [ 1.10140069  1.13537321  0.74833178  1.21907866  0.82567377]
 [ 0.72227738  0.54102632  0.41962657  1.98316579  1.73393467]]
(3, 5)
success!
```

Expected result: (*or similar*)\ [[1.17632192 1.32360047 1.83204165 -0.20224445 0.44039155]\ [1.10140069 1.13537321 0.74833178 1.21907866 0.82567377]\ [0.72227738 0.54102632 0.41962657 1.98316579 1.73393467]]\ \ (3, 5)

Exercise 1.4 (5 points)

Let's get your model to make predictions on the test data.

In [24]:
```
# Prediction on test data

if (X_test.shape[1] == X.shape[1]):
    X_test.insert(0, "intercept", 1)

# Reset m and n for test data
r,c = X_test.shape

y_pred = []
for index,row in X_test.iterrows(): # get a row of X_test data
    # calculate y_hat using hypothesis function
    y_hat = None
    # find the index (integer value) of maximum value in y_hat an
d input back to prediction
    prediction = None
    ### BEGIN SOLUTION
    y_hat = h(row, theta, num_class)
    prediction = int(np.where(y_hat == y_hat.max())[0])
    ### END SOLUTION
    # collect the result
    y_pred.append(prediction)
```

In [25]:
```
print(len(y_pred))
print(y_pred[:7])
print(type(y_pred[0]))

# Test function: Do not remove
assert len(y_pred) == 45, 'Size of y_pred is incorrect'
assert isinstance(y_pred[0], int) and isinstance(y_pred[15], int)
and isinstance(y_pred[17], int), 'prediction type is incorrect'
assert max(y_pred) < 3 and min(y_pred) >= 0, 'wrong index of y_pr
ed'

print("success!")
# End Test function
```

```
45
[0, 2, 1, 2, 1, 1, 0]
<class 'int'>
success!
```

Expected result: (*or similar*)\ 45 \ [2, 0, 2, 0, 0, 0, 2] \

<class 'int'>

Exercise 1.5 (5 points)

Estimate accuracy of model on test data

$$\text{accuracy} = \frac{\text{number of correct test predictions}}{m_{\text{test}}}$$

In [26]:
```
def calc_accuracy(y_test, y_pred):
    accuracy = None
    ### BEGIN SOLUTION
    m = len(y_test)
    correct = (y_pred == y_test).value_counts()[True]
    accuracy = correct/m
    ### END SOLUTION
    return accuracy
```

In [27]:
```
accuracy = calc_accuracy(y_test, y_pred)
print('Accuracy: %.4f' % accuracy)

# Test function: Do not remove
assert isinstance(accuracy, float), 'accuracy should be floating point'
assert accuracy >= 0.8, 'Did you train the data?'

print("success!")
# End Test function
```

```
Accuracy: 0.9333
success!
```

Expected result: should be at least 0.8!

On your own in lab

We will do the following in lab:

1. Write a function to obtain the cost for particular $\mathbf{X}$, $\mathbf{y}$, and θ.
2. Plot the training set and test cost as training goes on and find the best value for the number of iterations and learning rate.
3. Make 2D scatter plots showing the predicted and actual class of each item in the training set, plotting two features at a time. Comment on the cause of the errors you observe. If you obtain perfect test set accuracy, re-run the train/test split and rerun the optimization until you observe some mistaken predictions on the test set.

Exercise 2.1 (15 points)

1. Write a function to obtain the cost for particular $\mathbf{X}$, $\mathbf{y}$, and θ. Name your function `my_J()` and implement

$$J_j = -\delta(y, j) \log \phi_j$$

In [28]:
```
def my_J(theta, X, y, j, num_class):
    cost = None
    ### BEGIN SOLUTION
    cost = indicator(y,j)*-np.log(phi(j, theta, X, num_class))
    ### END SOLUTION
    return cost
```

In [29]:
```
# Test function: Do not remove
m, n = X_train.shape
test_theta = np.ones((3, n))
cost = my_J(test_theta, X_train.loc[10], y_train[10], 0, 3)
assert isinstance(cost, float), 'cost should be floating point'

print("success!")
# End Test function
```

```
success!
```

1. Implement `my_grad_cost` using your `my_J` function

In [30]:
```
def my_grad_cost(X, y, j, theta, num_class):
    grad = None
    cost = None
    ### BEGIN SOLUTION
    m, n = X.shape
    sum = np.array([0 for i in range(0,n)])
    cost = 0
    for i in range(0, m):
        p = indicator(y[i], j) - phi(j, theta,X.loc[i], num_clas
s)
        cost = cost + my_J(theta, X.loc[i], y[i], j, num_class)
        sum = sum + (X.loc[i] * p)
    grad = -sum/m
    ### END SOLUTION
    return grad, cost
```

In [31]:
```
# Test function: Do not remove
m, n = X_train.shape
test_theta = np.ones((3, n))
grad, cost = my_grad_cost(X_train, y_train, 0, test_theta, num_cl
ass)
print(grad)
print(cost)
assert isinstance(cost, float), 'cost should be floating point'
assert isinstance(grad['intercept'], float) and \
        isinstance(grad['SepalLengthCm'], float) and \
        isinstance(grad['SepalWidthCm'], float) and \
        isinstance(grad['PetalLengthCm'], float) and \
        isinstance(grad['PetalWidthCm'], float) , 'grad should be
floating point'
print("success!")
# End Test function
```

```
intercept       -0.009524
SepalLengthCm    0.243175
SepalWidthCm    -0.138730
PetalLengthCm    0.738095
PetalWidthCm     0.311746
dtype: float64
39.55004239205195
success!
```

Expect result: (*or similar*)\ intercept 0.009524\ SepalLengthCm 0.316825\ SepalWidthCm -0.091429\ PetalLengthCm 0.780000\ PetalWidthCm 0.329524\ dtype: float64\ 37.352817814715735

1. Implement `my_gradient_descent` using your `my_grad_cost` function

In [32]:
```
def my_gradient_descent(X, y, theta, alpha, iters, num_class):
    cost_arr = []
    ### BEGIN SOLUTION
    for iter in range(iters):
        cost = 0
        for j in range(0, num_class):
            grad = my_grad_cost(X, y, j, theta_initial, num_clas
s)
            theta[j] = theta[j] - alpha * grad[0]
            cost = cost + grad[1]
        cost_arr.append(cost)
    ### END SOLUTION
    return theta, cost_arr
```

In [33]:
```
# Test function: Do not remove
m, n = X_train.shape
test_theta = np.ones((3, n))
theta, cost = my_gradient_descent(X_train, y_train, theta_initia
l, 0.001, 5, 3)
print(theta)
print(cost)
print("success!")
# End Test function
```

```
[[1.00006092 0.99886592 1.00073352 0.99636928 0.99846126]
 [0.99995064 0.99980613 0.99937913 1.00061439 1.00013322]
 [0.99998402 1.00130089 0.99987409 1.00299674 1.00139899]]
[115.23455339572588, 115.10308521563118, 114.97425621563247, 114.
84796160531872, 114.7241007969092]
success!
```

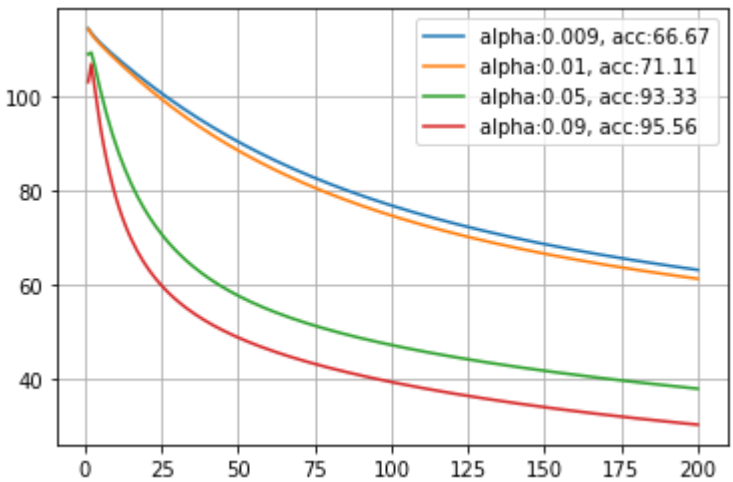
Expected result: (*or similar*)\ [[1.00001186 0.99618853 1.00183642 0.9889817 0.99528923]\ [1.00009697 1.0011823 0.99883395 1.00316763 1.00083055]\ [0.99987915 1.00255606 0.99929351 1.00779768 1.00386218]]\ [114.00099216453735, 113.89036233839263, 113.78163144339288, 113.67472269747496, 113.56956268162737]\ 37.352817814715735

Exercise 2.2 (20 points)

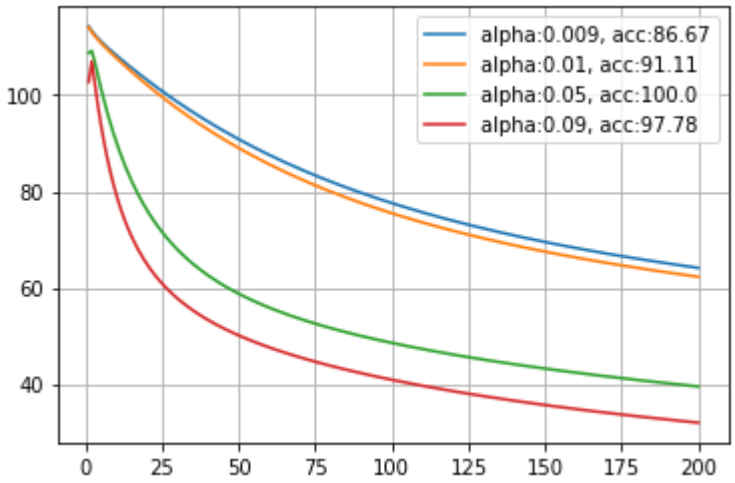
1. Plot the training set and test cost as training goes on and find the best value for the number of iterations and learning rate.
2. Make 2D scatter plots showing the predicted and actual class of each item in the training set, plotting two features at a time. Comment on the cause of the errors you observe. If you obtain perfect test set accuracy, re-run the train/test split and rerun the optimization until you observe some mistaken predictions on the test set.

In [34]:
```
import matplotlib.pyplot as plt
```

In [35]:
```
theta_arr = []
cost_arr = []
accuracy_arr = []

# design your own learning rate and num iterations
alpha_arr = np.array([None, None, None, None])
iterations_arr = np.array([None, None, None, None])

### BEGIN SOLUTION
alpha_arr = np.array([.009, .01, .05, .09])
iterations_arr = np.array([200, 200, 200, 200])


m, n = X_test.shape

fig = plt.figure()
ax = plt.axes()
plt.grid(axis='both')
for i in range(0, len(alpha_arr)):
    theta_initial = np.ones((num_class, n))
    theta, cost = my_gradient_descent(X_train, y_train, theta_ini
tial, alpha_arr[i], iterations_arr[i], num_class)

    y_pred = []
    for index,row in X_test.iterrows():
        h_matrix = h(row, theta, num_class)
        prediction = int(np.where(h_matrix == h_matrix.max())[0])
        y_pred.append(prediction)
    correct = (y_pred == y_test).value_counts()[True]
    accuracy = correct/m
    plt.plot(range(1,iterations_arr[i]+1), cost, label='alpha:'+s
tr(alpha_arr[i]) +', acc:' + str(np.round(accuracy,4)*100))
    accuracy_arr.append(accuracy)

plt.legend()
plt.show()
### END SOLUTION
```



Expected result: (*Yours doesn't have to be the same!*)



On your own to take home

We see that the Iris dataset is pretty easy. Depending on the train/test split, we get 95-100% accuracy.

Find a more interesting multi-class classification problem on Kaggle (Tell the reference), clean the dataset to obtain numerical input features without missing values, split the data into test and train, and experiment with multinomial logistic regression.

Write a brief report on your experiments and results. As always, turn in a Jupyter notebook by email to the instructor and TA.

In [ ]: