Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel$\rightarrow$Restart) and then **run all cells** (in the menubar, select Cell$\rightarrow$Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [ ]:
```
NAME = "Aung Zar Lin"
ID = "121956"
```

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 $\mathbf{\eta}$ with $\eta_i = \theta^\top_i \mathbf{x}$, 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 $$\phi_i = p(y = i \mid \mathbf{x}; \theta) = \frac{e^{\theta^\top_i \mathbf{x}}}{\sum_{j=1}^{K}e^{\theta^\top_j \mathbf{x}}}. $$

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

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:

$$\theta^* = \text{argmax}_\theta {\cal L}(\theta) = \text{argmax}_\theta \ell(\theta) = \text{argmin}_\theta J(\theta),$$
where $$\begin{eqnarray} J(\theta) & = & - \ell(\theta) \\ & = & - \sum_{i=1}^m \log p(y^{(i)} \mid \textbf{x}^{(i)} ; \theta). \end{eqnarray}$$

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: $$\begin{eqnarray} \frac{\partial J}{\partial \theta_{kl}} & = & - \sum_{i=1}^m \frac{1}{\phi_{y^{(i)}}} \frac{\partial \phi_{y^{(i)}}}{\partial \theta_{kl}}. \\ \end{eqnarray}$$ 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: $$ \frac{\partial a_j}{\partial \theta_{kl}} = \delta(k=j)x^{(i)}_l. $$ For the first factor, we have $$\begin{eqnarray} \frac{\partial \phi_{y^{(i)}}}{\partial a_j} & = & \frac{ \left[\delta(y^{(i)}=j)e^{a_j} \sum_{c=1}^K e^{a_c} \right] - e^{a_j} e^{a_j} }{\left[\sum_{c=1}^K e^{a_c} \right]^2} \\ & = & \delta(y^{(i)}=j) \phi_j - \phi_j^2 \end{eqnarray}$$

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 $$\frac{\partial a_j}{\partial \theta_{kl}} = \delta(j=k) x^{(i)}_l,$$ so we can simplify as follows: $$\begin{eqnarray} \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^{(i)}_l \\ & = & - \sum_{i=1}^m (\delta(y^{(i)}=k) - \phi_k) x^{(i)}_l \\ \end{eqnarray}$$

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

3. The cost function is $$J(\theta) = - \sum_{i=1}^m \log p(y^{(i)} \mid \textbf{x}^{(i)})$$

4. The optimization algorithm is gradient descent on $J(\theta)$ with the update rule $$\theta_{kl}^{(n+1)} \leftarrow \theta_{kl}^{(n)} - \alpha \sum_{i=1}^m (\delta(y^{(i)}=k) - \phi_k) x^{(i)}_l.$$

Multinomial Regression Example

The following example of multinomial logistic regression is from Kaggle.

The data set is the famous Iris dataset from the UCI machine learning repository.

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 [1]:
```
# importing libraries
import numpy as np
import pandas as pd
import random
import math
```

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

In [2]:
```
def phi(i, theta, X, num_class):
    """
    Here is how to make documentation for your function show up in 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 attributes
        - Press 'Shift+TAB' after typing a function name to see its documentation

    The `phi` function returns phi_i = h_theta(x) for input patterns X and parameters t

    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

In [ ]:

```
phi
```

```
Signature: phi(i, theta, X)
Docstring:
Here is how to make your function as intellisense. You can create function expla
nation in coding and it
will show you when you use it.

Open intellisens in jupyter notebook.
    - Press 'TAB' after dot(.)
    - Press 'Shift+TAB' after typing function done
```

In [3]:

```
def grad_cost(X, y, j, theta):
```

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

In [3]:

```
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 parameters 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_class)
        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 = theta_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 to learn how to use pandas.

In [4]:
```
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 = data[y_label]
    # 4. get index of y-column
    y_index = data.columns.get_loc(y_label)
    # 5. Extrack X features from data
    X = data.iloc[:, :y_index]
    # YOUR CODE HERE
    #raise NotImplementedError()
    return X, y
```

In [5]:
```
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, 'Incorrect 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.col
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 [6]:

```
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_labels_new
    # 7. Drop shuffle index columns
    #     - pandas.reset_index() and pandas.drop(...) might be help

    m = X.shape[0]
    idx = np.arange(0, m)
    random.shuffle(idx)
    m_train = int(percent_train * m)
    train_idx = idx[0:m_train]
    test_idx = idx[m_train:]

    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()
    y_labels_new = np.arange(len(y_labels_name))

    i = 0
    for label in y_labels_name:
        y_train[y_train.str.match(label)] = str(i)
        y_test[y_test.str.match(label)] = str(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).squeeze()

    y_test = y_test.reset_index()
    y_test = y_test.drop(['index'], axis=1).squeeze()

    # YOUR CODE HERE
    #raise NotImplementedError()

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

In [7]:

```
percent_train = 0.7
idx, X_train, y_train, X_test, y_test, y_labels_name, y_labels_new = partition(X, y, pe
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, 'number of y uniques are inc
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_labels_name and \
        'Iris-versicolor' in y_labels_name, 'y unique data incorrect'
assert min(y_labels_new) == 0 and max(y_labels_new) < 3, 'label indices 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            6.4           2.9            4.3           1.3
1            5.7           3.8            1.7           0.3
2            7.7           2.6            6.9           2.3
3            5.6           2.5            3.9           1.1
4            4.3           3.0            1.1           0.1
0    1
1    0
2    2
3    1
4    0
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 [8]:

```
# 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 = gradient_descent(X_train, y_train, theta_initial, alpha,iterations, num_class)
# Logistic regression
# YOUR CODE HERE
#raise NotImplementedError()
```

In [9]:

```
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
```

```
[[ 1.17126256  1.32631123  1.8318103  -0.20670386  0.45527533]
 [ 1.08571752  1.15725955  0.73902449  1.20164259  0.83623618]
 [ 0.74301993  0.51642922  0.42916521  2.00506128  1.70848849]]
(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 [10]:

```
# 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 = h(row, theta, num_class)
    # find the index (integer value) of maximum value in y_hat and input back to predic
    prediction = int(np.argmax(y_hat))
    # YOUR CODE HERE
    #raise NotImplementedError()
    # collect the result
    y_pred.append(prediction)
```

In [11]:

```
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
```

```
assert max(y_pred) < 3 and min(y_pred) >= 0, 'wrong index of y_pred'

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

```
45
[0, 1, 2, 2, 1, 0, 2]
<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 [12]:
```
def calc_accuracy(y_test, y_pred):
    accuracy = np.sum(y_test == y_pred) / y_test.shape[0]
    # YOUR CODE HERE
    #raise NotImplementedError()
    return accuracy
```

In [13]:
```
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: 0.9556
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 $\mathtt{X}$, $\mathbf{y}$, 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 $\mathtt{X}$, $\mathbf{y}$, and θ.
   Name your function `my_J()` and implement

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

In [14]:
```
def my_J(theta, X, y, j, num_class):
    cost = -(indicator(y,j))*(np.log(phi(j, theta, X, num_class)))
    # YOUR CODE HERE
    #raise NotImplementedError()
    return cost
```

In [15]:
```
# 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 [20]:
```
def my_grad_cost(X, y, j, theta, num_class):
    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_class)
        cost = cost + my_J(theta, X.loc[i], y[i], j, num_class)
        sum = sum + (X.loc[i] * p)
    grad = -sum/m
    # YOUR CODE HERE
    #raise NotImplementedError()
    return grad, cost
```

In [21]:
```
# 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_class)
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    0.228889
SepalWidthCm    -0.151429
PetalLengthCm    0.741587
PetalWidthCm     0.308254
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 [22]:
```
def my_gradient_descent(X, y, theta, alpha, iters, num_class):
    cost_arr = []
    for iter in range(iters):
        cost = 0
        for j in range(0, num_class):
            grad, my_cost = my_grad_cost(X, y, j, theta, num_class)
            theta[j] = theta[j] - alpha* grad
            cost = cost + my_cost
        cost_arr.append(cost)
    #raise NotImplementedError()
    return theta, cost_arr
```

In [23]:
```
# 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_initial, 0.001, 5, 3)
print(theta)
print(cost)
print("success!")
# End Test function
```

```
[[1.00006054 0.99893523 1.00079574 0.99635019 0.99847816]
 [0.99999589 1.00013946 0.99951164 1.00089665 1.00023796]
 [0.99993968 1.00090134 0.99968097 1.00273575 1.00127806]]
[115.24773005041051, 115.12647344506988, 115.00734875025574, 114.89026977546017, 114.775
15403210349]
success!
```

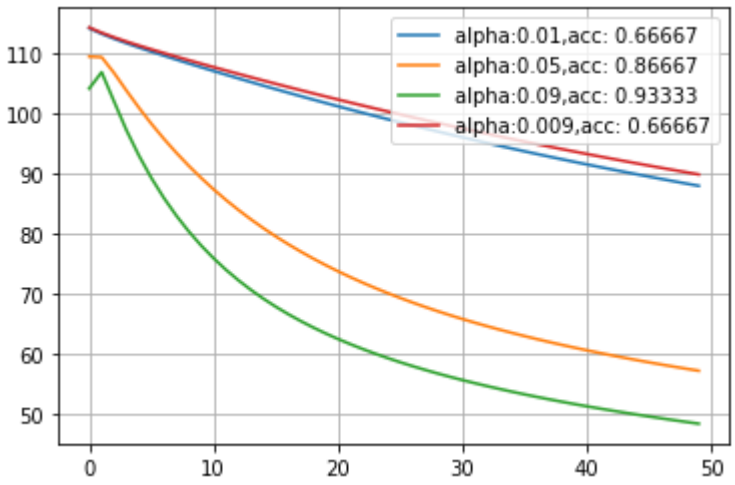
Expected result: (*or similar*)\ [[1.00001186 0.99618853 1.00183642 0.9889817 0.99528923]\ [1.00009697 1.0011823 0.99883395 1.00316763 1.00083055]\ [0.99987915 1.00255606 0.99929351 1.00779768 1.00386218]]\ [114.00099216453735, 113.89036233839263, 113.78163144339288, 113.67472269747496, 113.56956268162737]\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

Exercise 2.2 (20 points)

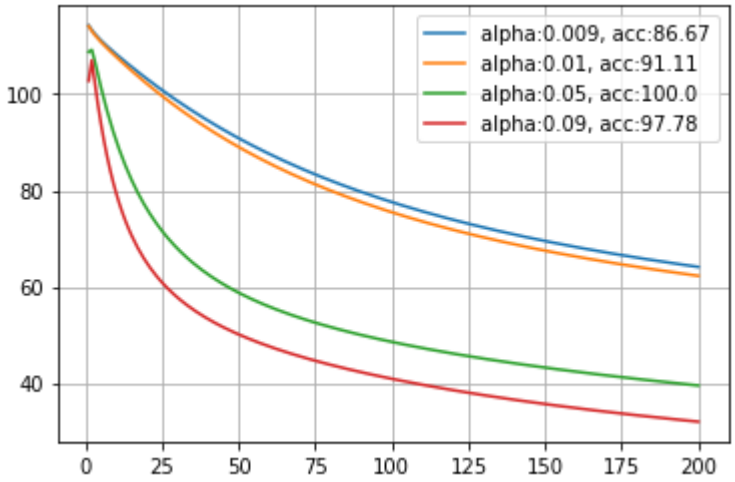
1. Plot the training set and test cost as training goes on and find the best value for the number of iterations and learning rate.
2. Make 2D scatter plots showing the predicted and actual class of each item in the training set, plotting two features at a time. Comment on the cause of the errors you 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 [24]:
```
import matplotlib.pyplot as plt
```

In [29]:
```
theta_arr = []
cost_arr = []
accuracy_arr = []

# design your own learning rate and num iterations
alpha_arr = np.array([0.01, 0.05, 0.09, 0.009])
iterations_arr = np.array([50, 50, 50, 50]) # reduce no of iter to finish faster

# YOUR CODE HERE
#raise NotImplementedError()

r, c = X_train.shape
num_class = len(y_labels_name)

for i in range(len(alpha_arr)):
    theta_initial = np.ones((num_class, c))
    theta, cost = my_gradient_descent(X_train, y_train, theta_initial, alpha_arr[i], it
    theta_arr.append(theta)
    cost_arr.append(cost)
    y_pred = []
    for index, row in X_test.iterrows():
        y_hat = h(row, theta, num_class)
        prediction = int(np.argmax(y_hat))
        y_pred.append(prediction)

    accuracy = calc_accuracy(y_test, y_pred)
    accuracy_arr.append(accuracy)

plt.grid()
for i in range(len(accuracy_arr)):
    plt.plot(range(iterations_arr[i]),cost_arr[i], label='alpha:'+str(alpha_arr[i])+ ',
    plt.legend(loc=1)
```



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 [30]:
```
def load_data(file_name, drop_label, is_print=False):

    data = pd.read_csv(file_name)

    if is_print:
        print(data.head())

    if drop_label is not None:
        data = data.drop(column=[drop_label], axis=1)
        if is_print:
            print(data.head())

    return data
```

In [31]:
```
data = load_data('Stars.csv', drop_label=None, is_print=True)
```

```
   Temperature         L       R    A_M Color Spectral_Class  Type
0         3068  0.002400  0.1700  16.12   Red              M     0
1         3042  0.000500  0.1542  16.60   Red              M     0
2         2600  0.000300  0.1020  18.70   Red              M     0
3         2800  0.000200  0.1600  16.65   Red              M     0
4         1939  0.000138  0.1030  20.06   Red              M     0
```

In [32]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 240 entries, 0 to 239
Data columns (total 7 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Temperature     240 non-null    int64
 1   L               240 non-null    float64
 2   R               240 non-null    float64
 3   A_M             240 non-null    float64
 4   Color           240 non-null    object
 5   Spectral_Class  240 non-null    object
 6   Type            240 non-null    int64
dtypes: float64(3), int64(2), object(2)
memory usage: 13.2+ KB
```

In [33]:

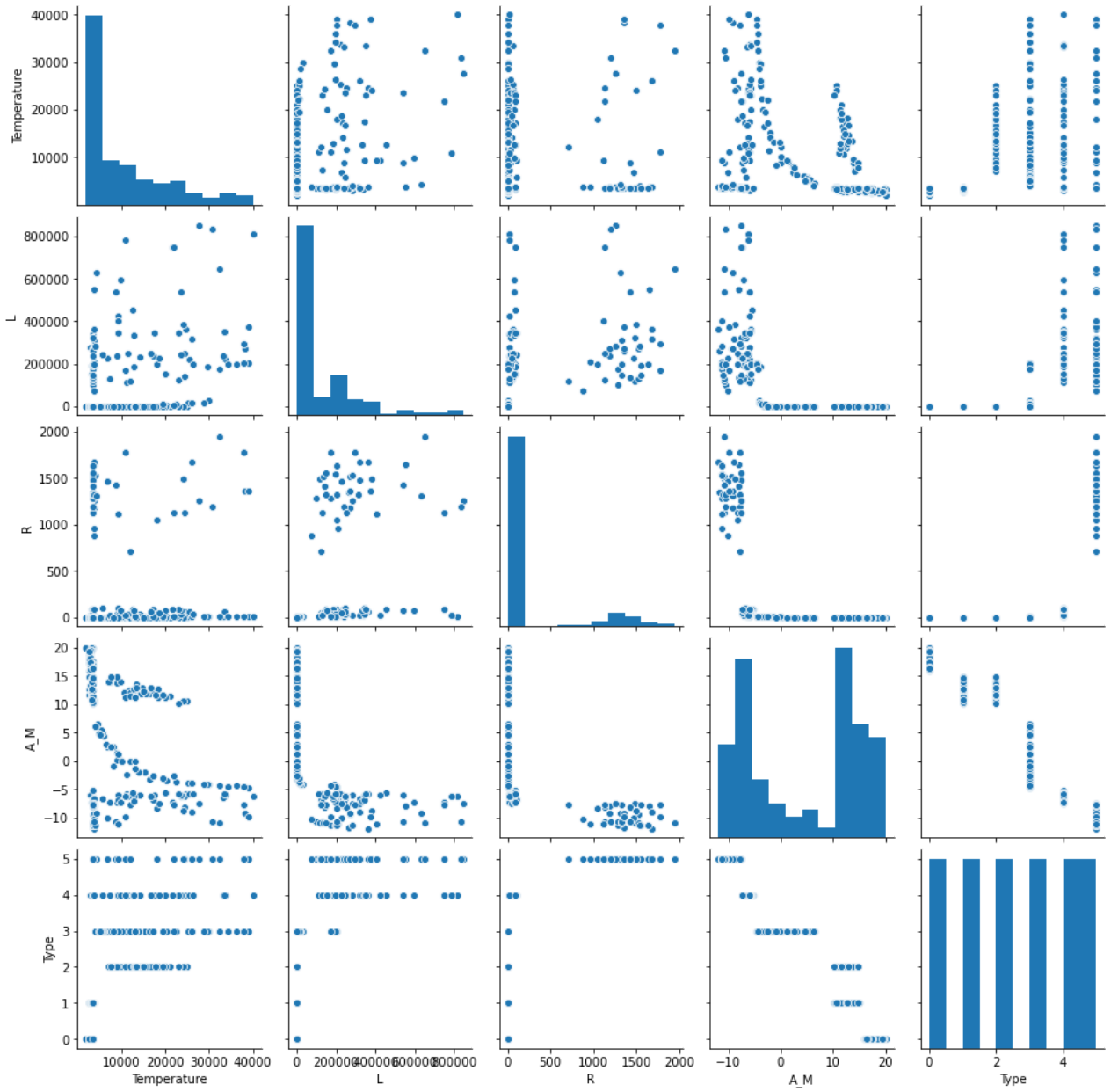
```
data.isnull().sum()
```

Out[33]:

```
Temperature       0
L                 0
R                 0
A_M               0
Color             0
Spectral_Class    0
Type              0
dtype: int64
```

In [34]:

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.pairplot(data)
plt.show()
```

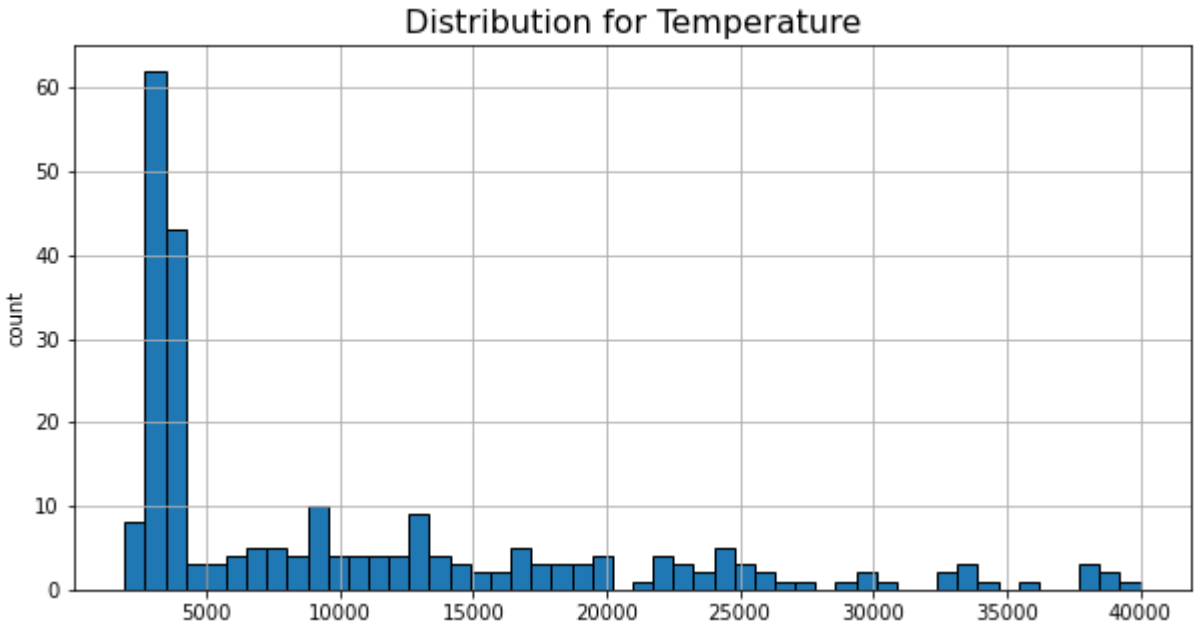


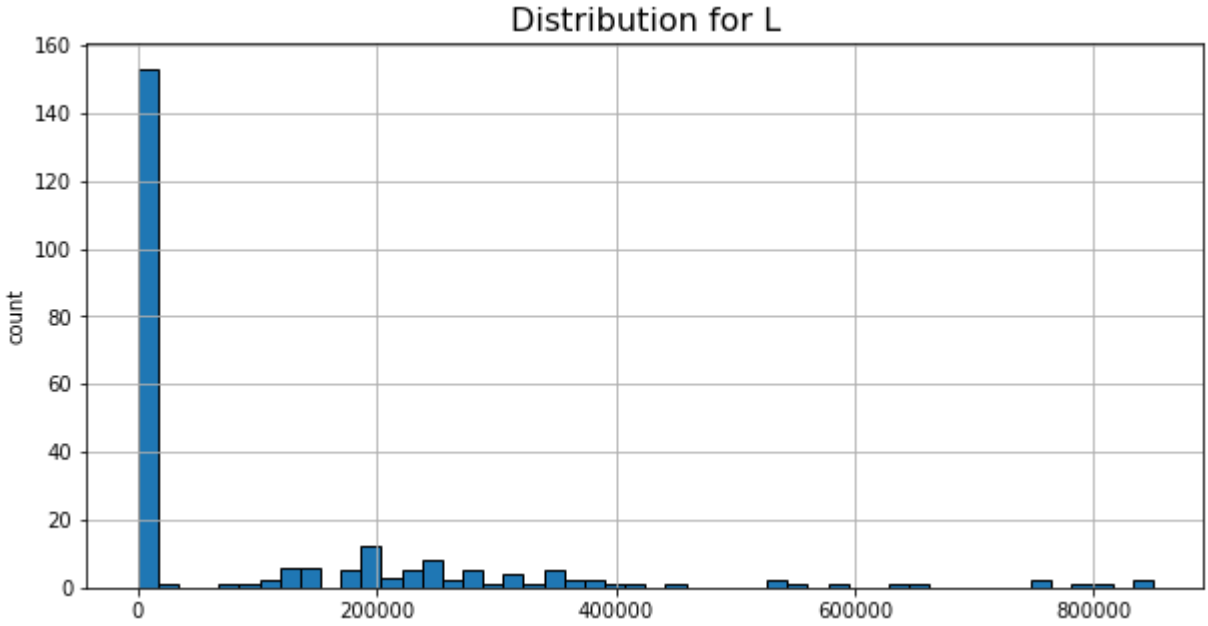
In [35]:
```
def histplot(data, name):

    plt.figure(figsize=(10,5))
    plt.grid()
    plt.hist(data[name], edgecolor='black', bins=50)
    plt.title(f"Distribution for {name}", size=16)
    plt.ylabel('count')
    plt.show()
```

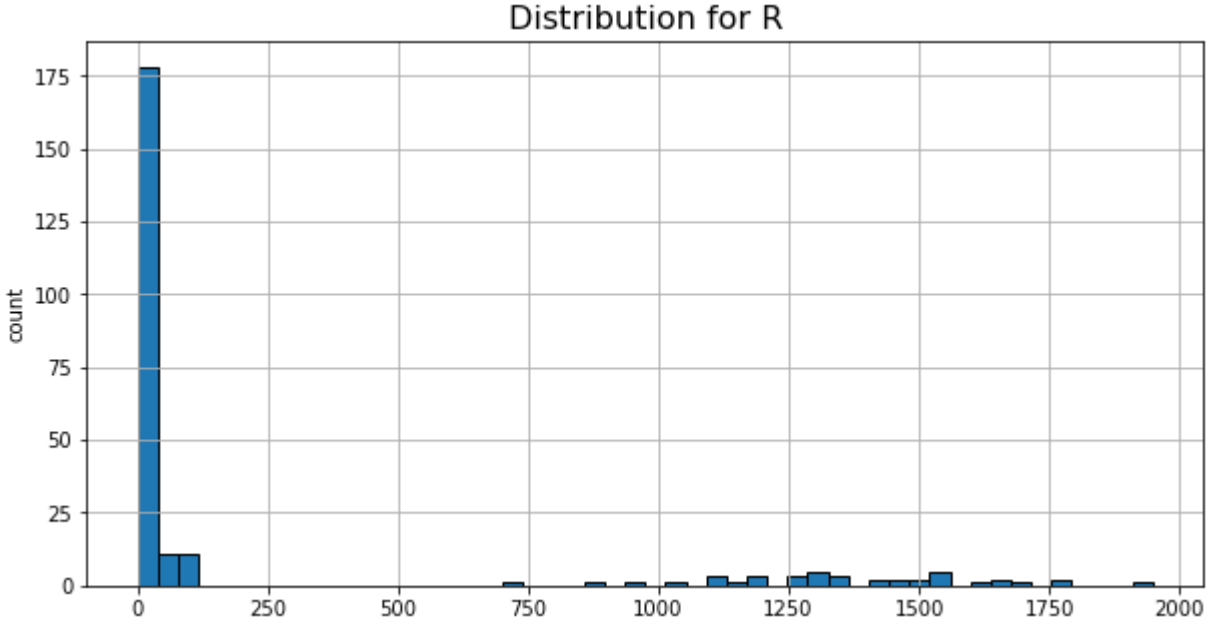
In [36]:
```
column_names = list(data.columns)
print(column_names)
```

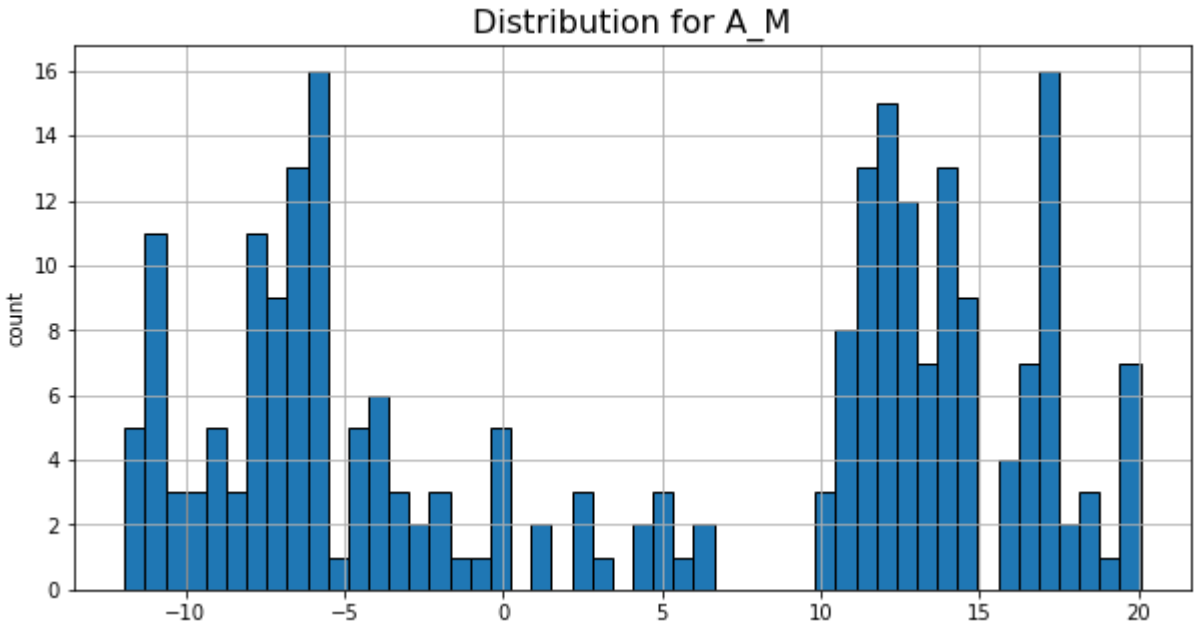
```
['Temperature', 'L', 'R', 'A_M', 'Color', 'Spectral_Class', 'Type']
```

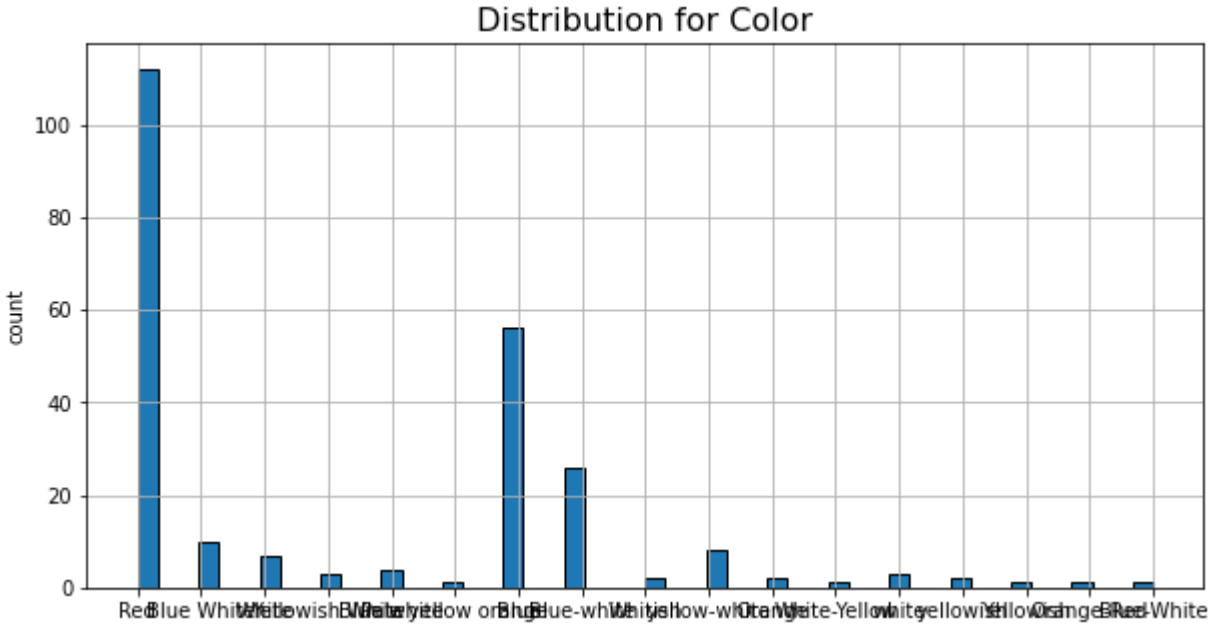
In [37]:
```
for name in column_names:
    histplot(data, name)
```

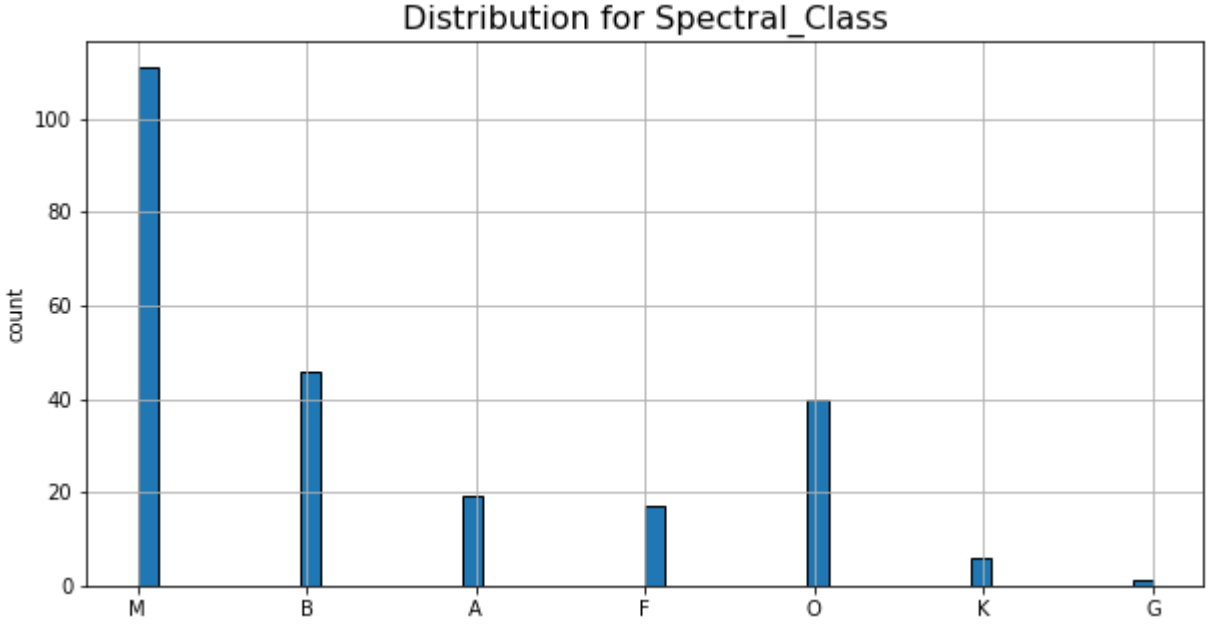

Distribution for Temperature
count
0
10
20
30
40
50
60
5000
10000
15000
20000
25000
30000
35000
40000

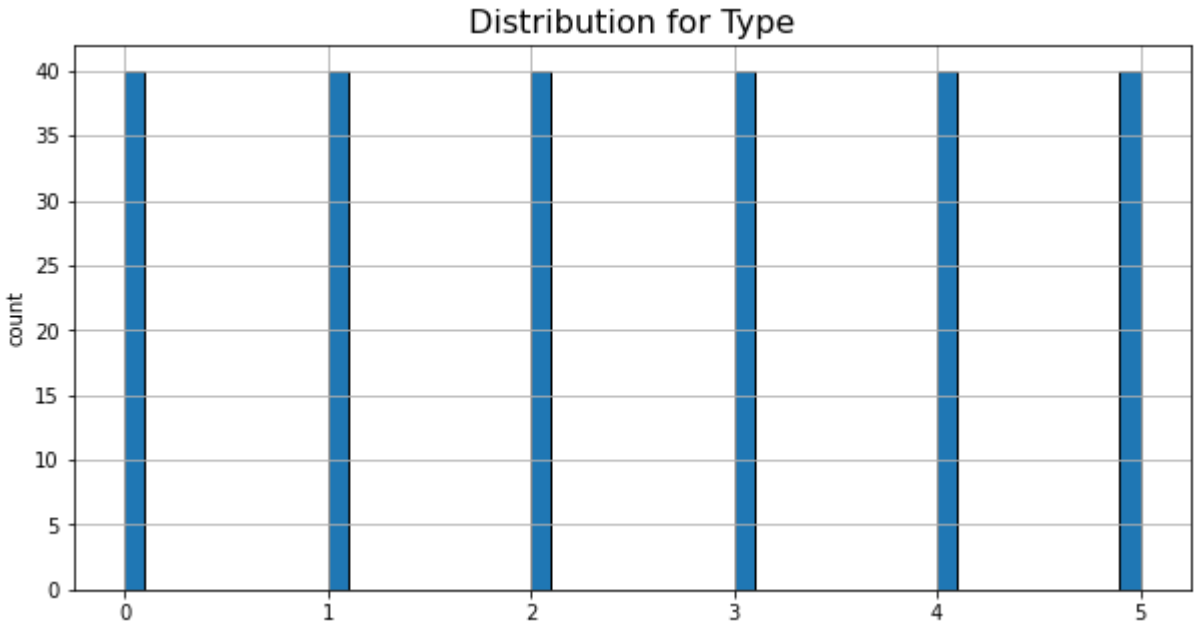

Distribution for L
count
0
20
40
60
80
100
120
140
160
0
200000
400000
600000
800000


Distribution for R
count
0
25
50
75
100
125
150
175
0
250
500
750
1000
1250
1500
1750
2000


Distribution for A_M
count
16
14
12
10
8
6
4
2
0
−10
−5
0
5
10
15
20


Distribution for Color
count
100
80
60
40
20
0


Distribution for Spectral_Class
count
100
80
60
40
20
0
M
B
A
F
O
K
G



In this datasets, Color column and Spectral_Class column has categorical values. We will change these values to numerical value.

In [38]:
```
print(data['Color'].unique())
print(len(data['Color'].unique()))
```

```
['Red' 'Blue White' 'White' 'Yellowish White' 'Blue white'
 'Pale yellow orange' 'Blue' 'Blue-white' 'Whitish' 'yellow-white'
 'Orange' 'White-Yellow' 'white' 'yellowish' 'Yellowish' 'Orange-Red'
 'Blue-White']
17
```

In [39]:
```
def cat_to_numeric(data, name):
    column_name = data[name].unique()
    column_new = np.arange(len(column_name))

    for i in range(len(column_name)):
        data[name] = data[name].replace(column_name[i], column_new[i])
```

In [40]:
```
cat_to_numeric(data, 'Color')
print(data['Color'].unique())
print(len(data['Color'].unique()))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16]
17
```

In [41]:
```
print(data['Spectral_Class'].unique())
print(len(data['Spectral_Class'].unique()))
```

```
['M' 'B' 'A' 'F' 'O' 'K' 'G']
7
```

In [42]:
```
cat_to_numeric(data, 'Spectral_Class')
print(data['Spectral_Class'].unique())
print(len(data['Spectral_Class'].unique()))
```

```
[0 1 2 3 4 5 6]
7
```

In [43]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 240 entries, 0 to 239
Data columns (total 7 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Temperature     240 non-null    int64  
 1   L               240 non-null    float64
 2   R               240 non-null    float64
 3   A_M             240 non-null    float64
 4   Color           240 non-null    int64  
 5   Spectral_Class  240 non-null    int64  
 6   Type            240 non-null    int64  
dtypes: float64(3), int64(4)
memory usage: 13.2 KB
```

In [44]:
```
y = data['Type']
y_index = data.columns.get_loc('Type')
X = data.iloc[:, :y_index]
print('X shape: ', X.shape)
print('Y shape: ', y.shape)
```

```
X shape:  (240, 6)
Y shape:  (240,)
```

In [45]:
```
def data_Norm(data):
    means = np.mean(data, axis=0)
    stds = np.std(data, axis=0)
    data_norm = (data - means) / stds
    return data_norm
```

In [46]:
```
X = data_Norm(X)
print(X.head())
```

```
   Temperature         L         R       A_M     Color  Spectral_Class
0    -0.779382 -0.598624 -0.459210  1.116745 -0.892015       -0.841613
1    -0.782110 -0.598624 -0.459241  1.162414 -0.892015       -0.841613
2    -0.828477 -0.598624 -0.459342  1.362213 -0.892015       -0.841613
3    -0.807496 -0.598624 -0.459229  1.167171 -0.892015       -0.841613
4    -0.897819 -0.598624 -0.459340  1.491607 -0.892015       -0.841613
```

In [47]:
```
def partition(X, y, percent_train):

    m = X.shape[0]
    idx = np.arange(m)
    random.shuffle(idx)
    m_train = int(m * percent_train)
    train_idx = idx[:m_train]
    test_idx = idx[m_train:]

    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 is already int
    y_labels_name = y.unique()
    #y_labels_new = np.arange(len(y_labels_name))

    #for i in range(len(y_labels_name)):
        #y_train = y_train.replace(y_labels_name[i], y_labels_new[i])
        #y_test = y_test.replace(y_labels_name[i], y_labels_new[i])

    #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).squeeze()

    y_test = y_test.reset_index()
    y_test = y_test.drop(['index'], axis=1).squeeze()

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

In [48]:

```
percent_train = 0.7
idx, X_train, X_test, y_train, y_test, y_labels_name = 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 lalel name:', y_labels_name)
print(X_train.head())
print(y_train.head())
```

```
X_train.shape (168, 6)
X_test.shape (72, 6)
y_train.shape (168,)
y_test.shape (72,)
y lalel name: [0 1 2 3 4 5]
   Temperature         L         R       A_M     Color  Spectral_Class
0    -0.765535  0.490409 -0.426599 -1.103881 -0.892015       -0.841613
1     0.253295 -0.598624 -0.459521  0.803726  0.181086       -0.231380
2     2.392083  0.741724 -0.436287 -1.035379  0.717636       -0.231380
3    -0.771619 -0.598624 -0.459039  0.926460 -0.892015       -0.841613
4    -0.738050 -0.598624 -0.459253  0.674333 -0.892015       -0.841613
0    4
1    2
2    4
3    1
4    1
Name: Type, dtype: int64
```

In [51]:

```
def phi(i, theta, X, num_class):
```

```
    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

def indicator(i, j):

    if i == j: return 1
    else: return 0


def grad_cost(X, y, j, theta, num_class):

    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_class)
        sum = sum + (X.loc[i] * p)
    grad = -sum / m
    return grad

def gradient_descent(X, y, theta, alpha, iters, num_class):

    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
        if iter % 10 == 0:
            print(f"Cost at iteration {iter}", cost)

    return theta

def h(X, theta, num_class):

    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

def my_J(theta, X, y, j, num_class):
    cost = -(indicator(y,j))*(np.log(phi(j, theta, X, num_class)))
    # YOUR CODE HERE
    #raise NotImplementedError()
    return cost

def my_grad_cost(X, y, j, theta, num_class):
    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_class)
        cost = cost + my_J(theta, X.loc[i], y[i], j, num_class)
        sum = sum + (X.loc[i] * p)
    grad = -sum/m
    # YOUR CODE HERE
    #raise NotImplementedError()
    return grad, cost

def my_gradient_descent(X, y, theta, alpha, iters, num_class):
    cost_arr = []
    # YOUR CODE HERE
    for iter in range(iters):
        cost = 0
        for j in range(0, num_class):
            grad, my_cost = my_grad_cost(X, y, j, theta, num_class)
            theta[j] = theta[j] - alpha* grad
            cost = cost + my_cost
        cost_arr.append(cost)
        if iter % 10 == 0:
            print(f"Cost at iteration {iter}", cost)
    #raise NotImplementedError()
    return theta, cost_arr

def calc_accuracy(y_test, y_pred):
    accuracy = np.sum(y_test == y_pred) / y_test.shape[0]
    # YOUR CODE HERE
    #raise NotImplementedError()
    return accuracy
```

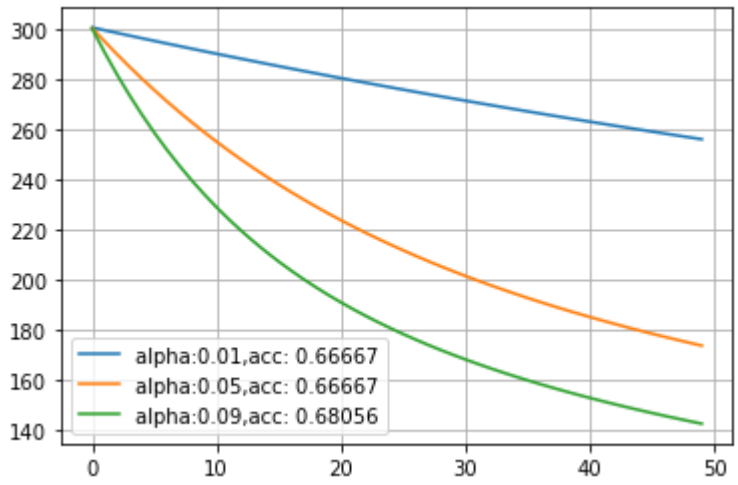
In [52]:

```
theta_arr = []
cost_arr = []
accuracy_arr = []

alpha_arr = np.array([0.01, 0.05, 0.09])
iterations_arr = np.array([50, 50, 50])

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

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

r, c = X_train.shape
num_class = len(y_labels_name)

for i in range(len(alpha_arr)):
    theta_initial = np.zeros((num_class, c))
    theta, cost = my_gradient_descent(X_train, y_train, theta_initial, alpha_arr[i], it
    theta_arr.append(theta)
    cost_arr.append(cost)
    y_pred = []
    for index, row in X_test.iterrows():
        y_hat = h(row, theta, num_class)
        prediction = int(np.argmax(y_hat))
        y_pred.append(prediction)

    accuracy = calc_accuracy(y_test, y_pred)
    accuracy_arr.append(accuracy)
```

```
Cost at iteration 0 300.95000860527057
Cost at iteration 10 290.4716159745265
Cost at iteration 20 280.748934782065
Cost at iteration 30 271.72876546979813
Cost at iteration 40 263.35926732355694
Cost at iteration 0 300.6949458614184
Cost at iteration 10 255.34167995690538
Cost at iteration 20 223.97421557459577
Cost at iteration 30 201.72117586752813
Cost at iteration 40 185.33172624019033
Cost at iteration 0 300.4513853470548
Cost at iteration 10 229.16024946279197
Cost at iteration 20 191.15315164013413
Cost at iteration 30 168.37415421674729
Cost at iteration 40 153.0162851817738
```

In [54]:

```
plt.grid()
for i in range(len(accuracy_arr)):
    plt.plot(range(iterations_arr[i]),cost_arr[i], label='alpha:'+str(alpha_arr[i])+ ',
    plt.legend()
```



SUMMARY

This is star classification problem. The data set has no null values but it has two categorical columns - Color and Spectral_Class. The categorical values in these two columns were changed into integer value (0, 1, 2, 3, ..). Each column in plotted to see the distribution. Next, the data was normalized and split into training set and test set with test size 0.3. The training set was trained with only three alpha values (0.01, 0.05, 0.09) with iterations 50. The number of iterations is only 50 to finish the loop faster but it still didn't reach to optimal theta value . As we can see from the graph, the cost is still high and it need atleast 1000 iterations.

dataseet:https://www.kaggle.com/brsdincer/star-type-classification

In [ ]: