Logistic Regression

Import necessary packages

```
import numpy as np
import matplotlib.pyplot as plt
import h5py # common package to interact with a dataset that is stored
on an H5 file.
import scipy
# from PIL import Image
from scipy import ndimage
```

Problem statement

Dataset ("data.h5") contains:

- a training set of m_train images labeled as cat (y=1) or non-cat (y=0)
- a test set of m_test images labeled as cat or non-cat
- each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB) (each image is square: height = num_px and width = num_px).

Task is to build a simple image-classification algorithm that can correctly classify pictures as cat or non-cat.

Overview dataset

Create shortcut for path

```
import os
cwd= os.getcwd() # current working directory
path = os.path.join(cwd,'data')
# print (path)
```

Load dataset

```
def load_dataset():
    file_name=  os.path.join(path , 'train_catvnoncat.h5')
    train_dataset = h5py.File(file_name, "r")
    X_train = np.array(train_dataset["train_set_x"][:]) # your train
set features
    Y_train = np.array(train_dataset["train_set_y"][:]) # your train
set labels

    file_name=  os.path.join(path , 'test_catvnoncat.h5')
    test_dataset = h5py.File(file_name, "r")
    X_test = np.array(test_dataset["test_set_x"][:]) # your test set
features
    Y_test = np.array(test_dataset["test_set_y"][:]) # your test set
labels

    classes = ['non-cat','cat']

    Y_train = Y_train.reshape(-1,1)
    Y_test = Y_test.reshape(-1,1)

    return X_train, Y_train, X_test, Y_test, classes

X_train,Y_train, X_test, Y_test, classes = load_dataset()
# Note : in case file not found, uncomment to print path in previous
step and correct to necessary exrension

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)
```

```
X_train.shape=  (209, 64, 64, 3)
X_test.shape=  (50, 64, 64, 3)
Y_train.shape=  (209, 1)
Y_test.shape=  (50, 1)
```

```
# YOUR_CODE.   get m_train, num_px and m_test
# START_CODE
m_train = X_train.shape[0]
```

```
num_px = X_train.shape[1]
m_test = X_test.shape[0]
# END_CODE
```

Check result

```
print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) +
", 3)")
```

```
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
Each image is of size: (64, 64, 3)
```

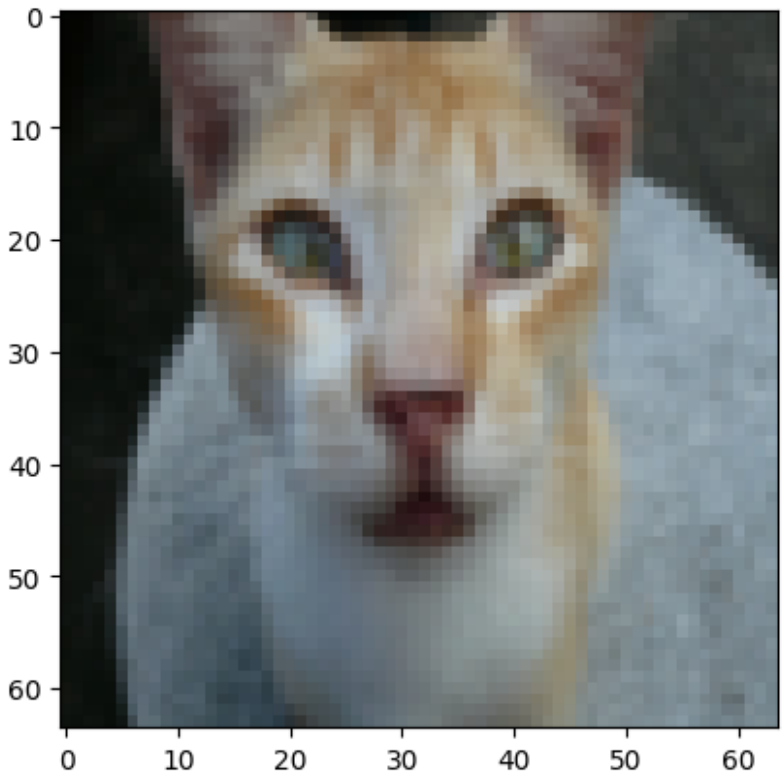
Expected output

`Number of training examples: m_train = 209 Number of testing examples: m_test = 50 Height/Width of each image: num_px = 64 Each image is of size: (64, 64, 3)`

Review some examples

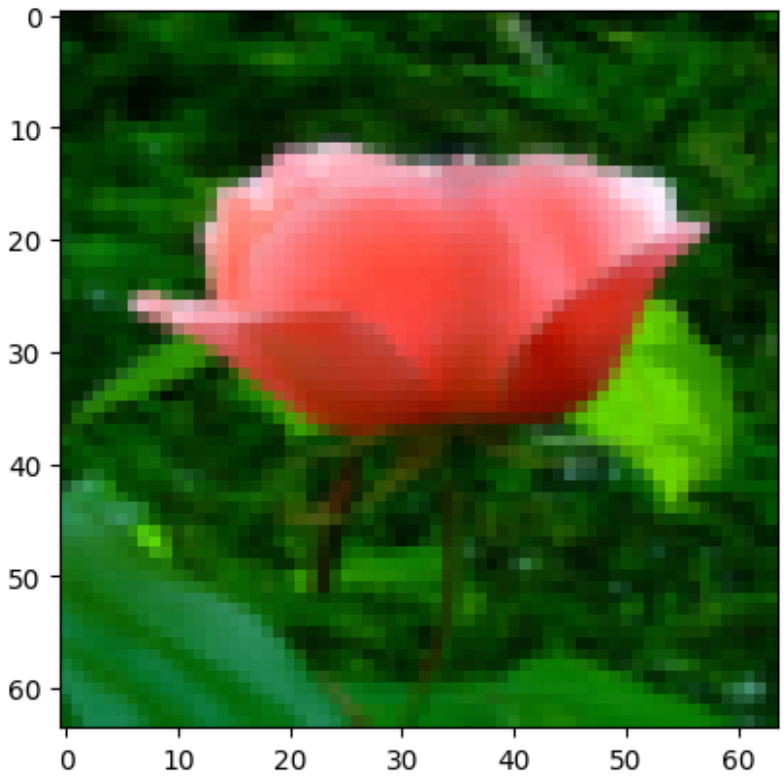
```
# Example of a picture
index = 11
plt.imshow(X_train[index])
print ("y = " + str(Y_train[index,:]) + ", it's a '" +
classes[np.squeeze(Y_train[index,:])] +  "' picture.")
```

```
y = [1], it's a 'cat' picture.
```




```
index = 3
plt.imshow(X_train[index])
print ("y = " + str(Y_train[index,:]) + ", it's a '" +
classes[np.squeeze(Y_train[index,:])] +  "' picture.")
```

```
y = [0], it's a 'non-cat' picture.
```




```
print ('Label 1 count:', np.sum(Y_train!=0))
print ('Label 0 count:', np.sum(Y_train==0))
```

```
Label 1 count: 72
Label 0 count: 137
```

Flatten features

```
# YOUR_CODE. Reshape the training and test set to shape
(number_of_samples,  num_px*num_px*3)
# START_CODE
X_train_flatten = X_train.reshape(m_train, -1)
X_test_flatten = X_test.reshape(m_test, -1)
# END_CODE
```

Check result

```
print ("train_set_x_flatten shape: {}".format(X_train_flatten.shape))
print ("test_set_x_flatten shape: {}".format(X_test_flatten.shape))
print ("sanity check after reshaping:
{}".format(X_train_flatten[0, :5]))
```

```
train_set_x_flatten shape: (209, 12288)
test_set_x_flatten shape: (50, 12288)
sanity check after reshaping: [17 31 56 22 33]
```

Expected Output

```
X_train_flattern shape: (209,12288) X_test_flattern shape: (50,12288)
sanity check after reshaping: [17 31 56 22 33]
```

Normalize features

To represent color images, the red, green and blue channels (RGB) must be specified for each pixel, and so the pixel value is actually a vector of three numbers ranging from 0 to 255.

For picture datasets, it is almost the same as MinMaxScaler to just divide every row of the dataset by 255 (the maximum value of a pixel channel).

```
X_train_scaled = X_train_flatten/255.
X_test_scaled = X_test_flatten/255.
```

sigmoid function

$ g(z) = \frac{1}{1 + e^{-z}}$

```
def sigmoid(z):
    """
    Compute the sigmoid of z

    Arguments:
    z -- A scalar or numpy array of any size.

    Return:
    g -- sigmoid(z)
    """
```

```
    # YOUR_CODE. Implement sigmoid function
    # START_CODE
    g = 1/(1+np.exp(-z))
    # END_CODE

    return g
```

Check result

```
print ("sigmoid([0, 2]) = " + str(sigmoid(np.array([0,2]))))
```

```
sigmoid([0, 2]) = [0.5         0.88079708]
```

Expected Output

```
sigmoid([0, 2]) [0.5         0.88079708]
```

Initialize parameters

```
def initialize_with_zeros(dim):
    """
    This function creates a vector of zeros of shape (1,dim) for w and
initializes b to 0.

    Argument:
    dim -- size of the w vector we want (or number of parameters in
this case)

    Returns:
    w -- initialized vector of shape (1,dim)
    b -- initialized scalar (corresponds to the bias)
    """

    # YOUR_CODE.  Initialize b to zero and w as a vector of zeros.
    # START_CODE
    w = np.zeros((1, dim))
```

```
    b = 0
    # END_CODE

    assert(w.shape == (1,dim))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b
```

Check result

```
dim = 2
w, b = initialize_with_zeros(dim)
print ("w = " + str(w))
print ("b = " + str(b))
```

```
w = [[0. 0.]]
b = 0
```

Expected Output

```
    w =  [[ 0.  0.]]
    b = 0
```

Note: For image inputs, w will be of shape (1, num_px × num_px × 3).

Forward and Backward propagation

Computing cost function is called "forward" and computing derivatives (gradient) is called "backward" propagation

Forward Propagation:

- compute $Z=b+X@w.T\left(\left[\left[z^{(0)}\right],\left[z^{(1)}\right],\ldots,\left[z^{(m-1)}\right]\right]\right)$
- compute $A=g\left(Z\right)\left(\left[\left[a^{(0)}\right],\left[a^{(1)}\right],\ldots,\left[a^{(m-1)}\right]\right]\right)$
- calculate the cost function: $J=-\frac{1}{m}\sum_{i=0}^{m-1}\left(y^{(i)}\log\left(a^{(i)}\right)+\left(1-Y^{(i)}\right)\log\left(1-a^{(i)}\right)\right)$

Backward Propagation:

$$\frac{\partial J}{\partial w} = \frac{1}{m}(A - Y)^T @ X \quad \frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(a^{(i)} - y^{(i)}\right)$$

```
# GRADED FUNCTION: propagate

def propagate(w, b, X, Y, C=1):
    """
    Implement the cost function and its gradient for the propagation
explained above

    Arguments:
    w -- weights, a numpy array of size (1,num_px * num_px * 3)
    b -- bias, a scalar
    X -- data of size (number of examples, num_px * num_px * 3)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of
size (number of examples,1)

    Return:
    cost -- negative log-likelihood cost for logistic regression
    dw -- gradient of the loss with respect to w, thus same shape as w
    db -- gradient of the loss with respect to b, thus same shape as b

    """
    m = X.shape[0]

    # YOUR_CODE.  implement forward propagation
    # START_CODE
    Z = b + X @ w.T
    A = sigmoid(Z)
    cost = (-1 / m) * np.sum(Y * np.log(A) + (1 - Y) * np.log(1 - A))
+ 1 / (2 * m) * np.sum(w ** 2)

    # END_CODE

    # YOUR_CODE.  Implement Backward propahation
    # START_CODE
    dJ_dw = (1 / m) * (A - Y).T @ X + 1 / m * w
    dJ_db = (1 / m) * np.sum(A - Y)

    # END_CODE

    assert (dJ_dw.shape == w.shape)
    assert (dJ_db.dtype == float)
    assert (cost.dtype == float)

    grads = {"dJ_dw": dJ_dw,
             "dJ_db": dJ_db}
```

```
    return grads, cost
```

Check result

```
w, b, X, Y = np.array([[1., 2.]]), 2., np.array([[1.,2.,-1.],[3.,4.,-3.2]]).T, np.array([[1,0,1]]).T
grads, cost = propagate(w, b, X, Y)
print ("dJ_dw = " + str(grads["dJ_dw"]))
print ("dJ_db = " + str(grads["dJ_db"]))
print ("cost = " + str(cost))

dJ_dw = [[1.33178935 3.06173906]]
dJ_db = 0.001455578136784208
cost = 6.6348786527278865
```

Expected Output

```
dJ_dw = [[1.33178935 3.06173906]]
dJ_db = 0.001455578136784208
cost = 6.6348786527278865
```

Optimization

```
def optimize(w, b, X, Y, num_iterations, learning_rate, C= 1, verbose = False):
    """
    This function optimizes w and b by running a gradient descent algorithm

    Arguments:
    w -- weights, a numpy array of size (1,num_px * num_px * 3)
    b -- bias, a scalar
    X -- data of size (number of examples, num_px * num_px * 3)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (number of examples,1)
    num_iterations -- number of iterations of the optimization loop
```

```
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- True to print the loss every 100 steps

    Returns:
    params -- dictionary containing the weights w and bias b
    grads -- dictionary containing the gradients of the weights and
bias with respect to the cost function
    costs -- list of all the costs computed during the optimization,
this will be used to plot the learning curve.
    """

    costs = [] # keep history for plotting if necessary

    for i in range(num_iterations):


        # YOUR_CODE.  Call to compute cost and gradient
        # START_CODE
        grads, cost = propagate(w, b, X, Y, C=1)
        # END_CODE

        # Retrieve derivatives from grads
        dJ_dw = grads["dJ_dw"]
        dJ_db = grads["dJ_db"]

        # YOUR_CODE.  Update paramaters
        # START_CODE
        w = w-learning_rate*dJ_dw
        b = b-learning_rate*dJ_db
        # END_CODE

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)

        # Print the cost every 100 training iterations
        if verbose and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    params = {"w": w,
              "b": b}

    grads = {"dJ_dw": dJ_dw,
             "dJ_db": dJ_db}

    return params, grads, costs
```

Check result

```
params, grads, costs = optimize(w, b, X, Y, num_iterations= 100,
learning_rate = 0.009, verbose = False)

print ("w = " + str(params["w"]))
print ("b = " + str(params["b"]))
print ("dw = " + str(grads["dJ_dw"]))
print ("db = " + str(grads["dJ_db"]))
```

```
w = [[ 0.08006006 -0.02399336]]
b = 1.9060971483059892
dw = [[0.62090316 1.19256883]]
db = 0.2084129285706479
```

Expected Output

```
w = [[ 0.08006006 -0.02399336]]
b = 1.9060971483059892
dw = [[0.62090316 1.19256883]]
db = 0.2084129285706479
```

Predict

1. Calculate $\hat{Y} = A = g(b + X @ w.T)$
2. Convert the entries of a into 0 (if activation <= 0.5) or 1 (if activation > 0.5), store the predictions in a vector `Y_prediction`. Try to avoid `for loop` but use vectorized way if possible.

```
def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned logistic
regression parameters (w, b)

    Arguments:
    w - weights, a numpy array of size (1,num_px * num_px * 3)
    b - bias, a scalar
    X - data of size (number of examples, num_px * num_px * 3)
```

```
    Returns:
    Y_prediction - a numpy array of shape (number of examples, 1)
containing all predictions (0/1) for the examples in X
    '''
    m,n = X.shape
    assert (w.shape==(1,n))

    # YOUR_CODE.  Compute "A" predicting the probabilities of a cat
being present in the picture
    # START_CODE
    A= sigmoid(b + X @ w.T)
    # END_CODE

    # YOUR_CODE.  Convert probabilities to actual predictions 0 or 1
    # START_CODE
    Y_prediction = (A >= 0.5).astype(int)
    # END_CODE

    assert(Y_prediction.shape == (m, 1))

    return Y_prediction
```

Check result

```
w = np.array([[0.1124579],[0.23106775]]).T
b = -0.3
X = np.array([[1.,-1.1,-3.2],[1.2,2.,0.1]]).T
print ("predictions = \n{}".format (predict(w, b, X)))
```

```
predictions =
[[1]
 [1]
 [0]]
```

Expected Output

```
predictions= [[1] [1] [0]]
```

Model

```
def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000,
learning_rate = 0.5, verbose = False, C= 1):
    """
    Builds the logistic regression model by calling the functions
implemented previously

    Arguments:
    X_train -- training set represented by a numpy array of shape
(number of examples, num_px * num_px * 3)
    Y_train -- training labels represented by a numpy array (vector)
of shape (1, m_train)
    X_test -- test set represented by a numpy array of shape (num_px *
num_px * 3, m_test)
    Y_test -- test labels represented by a numpy array (vector) of
shape (number of examples,1)
    num_iterations -- hyperparameter representing the number of
iterations to optimize the parameters
    learning_rate -- hyperparameter representing the learning rate
used in the update rule of optimize()
    print_cost -- Set to true to print the cost every 100 iterations
    C- regularization parameter

    Returns:
    res -- dictionary containing information about the model.
    """

    # YOUR_CODE.
    # START_CODE

    #  initialize parameters
    dim = X_train.shape[1]
    w, b = initialize_with_zeros(dim)

    # run gradient descent
    parameters, grads, costs = optimize(w, b, X_train, Y_train,
num_iterations, learning_rate, C= 1, verbose=verbose)


    # retrieve parameters w and b from dictionary "parameters"
    w = parameters.get('w')
    b = parameters.get('b')

    # predict test/train set examples
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)
```

```
    # END_CODE

    # Print train/test Errors
    print("train accuracy= {:.3%}".format(np.mean(Y_prediction_train
== Y_train)))
    print("test accuracy= {:.3%}".format(np.mean(Y_prediction_test ==
Y_test)))

    res = {'costs': costs,
           'Y_prediction_test': Y_prediction_test,
           'Y_prediction_train' : Y_prediction_train,
           'w' : w,
           'b' : b,
           'learning_rate' : learning_rate,
           'num_iterations': num_iterations,
           'C':C
          }

    return res
```

Check result

```
res = model(X_train= X_train_scaled,
            Y_train=Y_train,
            X_test=X_test_scaled,
            Y_test= Y_test,
            num_iterations = 3000,
            learning_rate = 0.005,
            verbose = True,
            C= 0.3 # 0.6 is still overfitting,   0.3  is low value to
keep the test accuracy ashigh as possible
           )
```

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584911
Cost after iteration 200: 0.468157
Cost after iteration 300: 0.377985
Cost after iteration 400: 0.333494
Cost after iteration 500: 0.305956
Cost after iteration 600: 0.283224
Cost after iteration 700: 0.264051
Cost after iteration 800: 0.247613
Cost after iteration 900: 0.233335
Cost after iteration 1000: 0.220802
Cost after iteration 1100: 0.209705
```

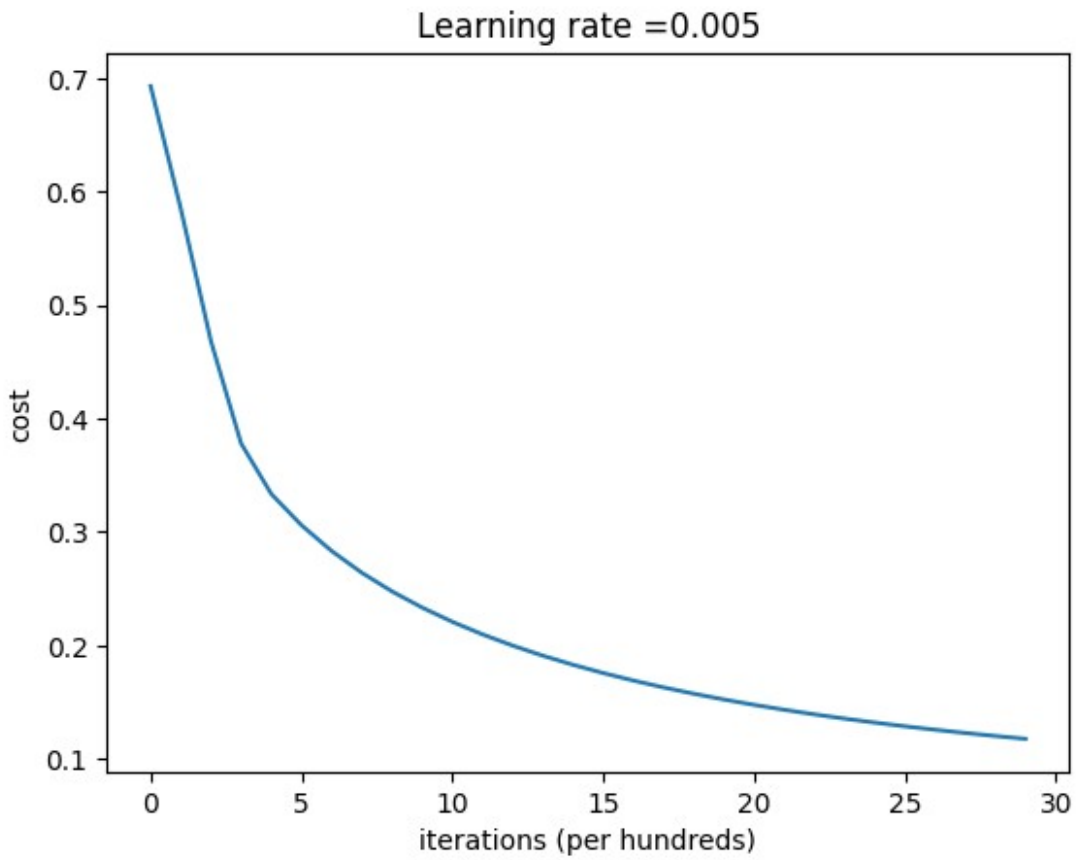
```
Cost after iteration 1200: 0.199806
Cost after iteration 1300: 0.190921
Cost after iteration 1400: 0.182902
Cost after iteration 1500: 0.175630
Cost after iteration 1600: 0.169008
Cost after iteration 1700: 0.162955
Cost after iteration 1800: 0.157403

Cost after iteration 1900: 0.152296
Cost after iteration 2000: 0.147584
Cost after iteration 2100: 0.143226
Cost after iteration 2200: 0.139186
Cost after iteration 2300: 0.135433
Cost after iteration 2400: 0.131939
Cost after iteration 2500: 0.128679
Cost after iteration 2600: 0.125634
Cost after iteration 2700: 0.122783
Cost after iteration 2800: 0.120111
Cost after iteration 2900: 0.117603
train accuracy= 99.522%
test accuracy= 68.000%
```

Expected Output

```
...
Cost after iteration 2600: 0.057748
Cost after iteration 2700: 0.056804
Cost after iteration 2800: 0.055918
Cost after iteration 2900: 0.055084
train accuracy= 98.565%
test accuracy= 70.000%
```

Visualize cost function changes

```
costs = np.squeeze(res['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(res["learning_rate"]))
plt.show()
```



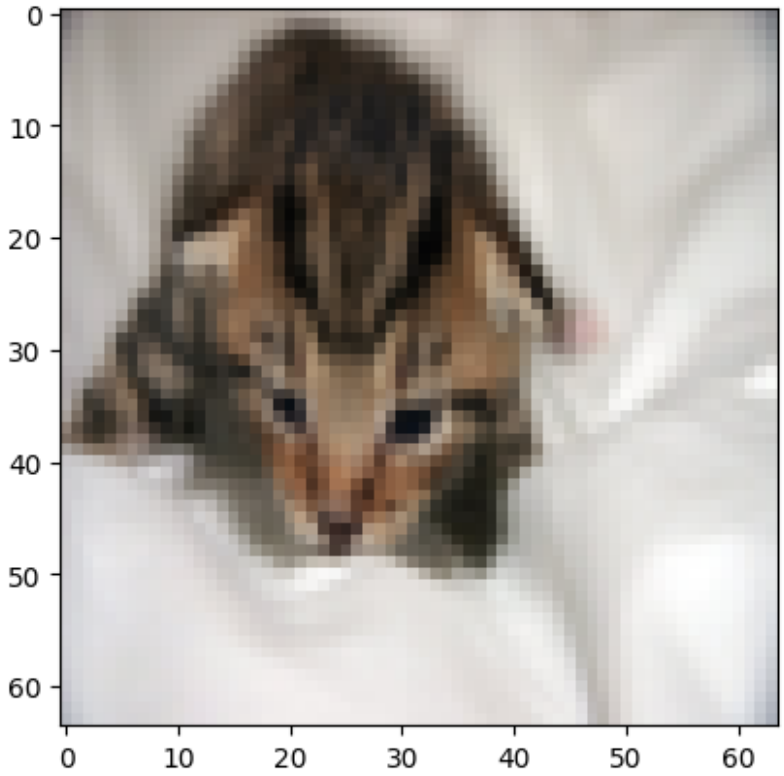
Visualize prediction

```
Y_test[index,0], res["Y_prediction_test"][index,0]
```

```
(1, 1)
```

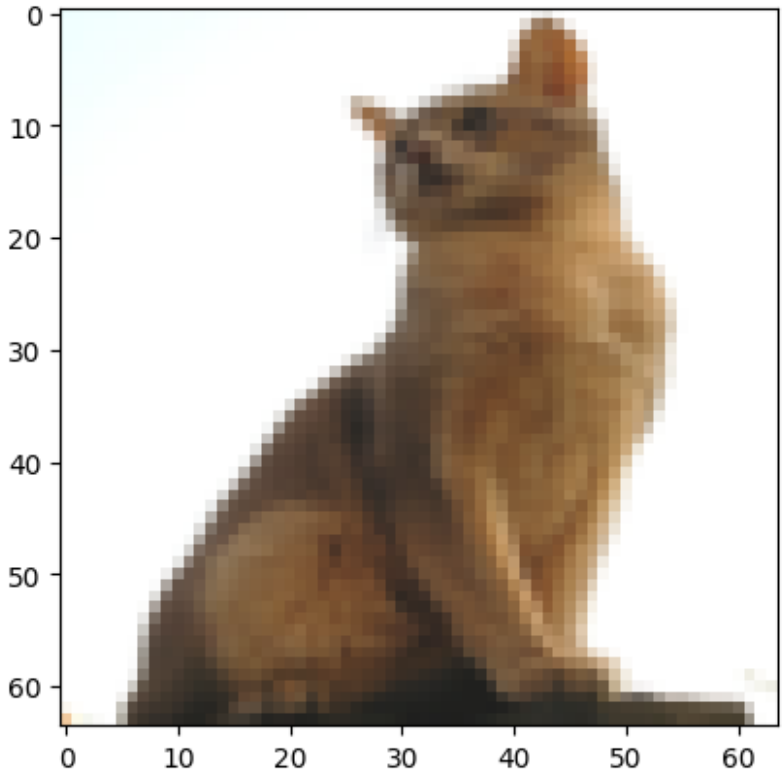
```
index = 1

plt.imshow(X_test[index,:].reshape(num_px, num_px, 3))
y_true = Y_test[index,0]
y_predicted =  res["Y_prediction_test"][index,0]
print ('y_predicted = {} (true label = {}) , you predicted that it is
a {} picture.'.\
       format(y_predicted,
              y_true,
              classes[y_predicted]))
# todo it looks like it does not use predicted value but test label
instead
```

```
y_predicted = 1 (true label = 1) , you predicted that it is a cat
picture.
```




```
# index = 6 is example of a picture that was wrongly classified.
index = 6
plt.imshow(X_test[index,:].reshape(num_px, num_px, 3))
y_true = Y_test[index,0]
y_predicted =  res["Y_prediction_test"][index,0]
print ('y_predicted = {} (true label = {}) , you predicted that it is
a {} picture.'.\
       format(y_predicted,
              y_true,
              classes[y_predicted]))
# todo it looks like it does not use predicted value but test label
instead
```

```
y_predicted = 0 (true label = 1) , you predicted that it is a non-cat
picture.
```




```
Y_test[index,0]
```

```
1
```

Test with your image

```
from PIL import Image
file_name=  os.path.join(path , 'Oleksiy.Tsebriy.jpg')
image = Image.open(file_name).resize((num_px,num_px))
print ('image.size= ', image.size)
image
```

```
image.size=  (64, 64)
```

```
my_image= np.array(image.getdata())
my_image.shape
```

```
(4096, 3)
```

```
my_image= my_image.reshape((1, num_px*num_px*3))
print ('my_image.shape=',my_image.shape)
my_image
```

```
my_image.shape= (1, 12288)
```

```
array([[227, 227, 217, ...,  43,  65,  96]])
```

```
import warnings
warnings.filterwarnings("ignore")

my_predicted_image = predict(res["w"], res["b"], my_image)
my_predicted_image
print('y = {} , your algorithm predicts a {} picture.'.

format(np.squeeze(my_predicted_image),classes[np.squeeze(my_predicted_
image)]))

warnings.filterwarnings("default")
```

```
y = 0 , your algorithm predicts a non-cat picture.
```

```
file_name=  os.path.join(path , 'test_cat.jpg')
image = Image.open(file_name).resize((num_px,num_px))
print ('image.size= ', image.size)
image = image.convert('RGB')
image
```

```
image.size=  (64, 64)
```

```
my_image= np.array(image.getdata())[:,:3] # by unknown reason picture
made as screenshot has 4 channels
print ('my_image.shape=', my_image.shape)
my_image= my_image.reshape((1, num_px*num_px*3))
print ('after reshape: my_image.shape=',my_image.shape)
my_predicted_image = predict(res["w"], res["b"], my_image)

print('y = {} , your algorithm predicts a {} picture.'.
```

```
format(np.squeeze(my_predicted_image),classes[np.squeeze(my_predicted_
image)]))

my_image.shape= (4096, 3)
after reshape: my_image.shape= (1, 12288)
y = 1 , your algorithm predicts a cat picture.
```

Sklearn implementation

```
from sklearn.linear_model import LogisticRegression

y_train = np.squeeze(Y_train) # LogisticRegression requires 1d input
for y
clf = LogisticRegression(C=0.01).fit(X_train_scaled, y_train)

print("train accuracy= {:.3%}".format(clf.score (X_train_scaled,
y_train)))
y_test = np.squeeze(Y_test)
print("test accuracy= {:.3%}".format(clf.score (X_test_scaled,
y_test)))

train accuracy= 90.909%
test accuracy= 66.000%

print('y = {} , sklearn algorithm predicts a {} picture.'.

format(np.squeeze(clf.predict(my_image)),classes[np.squeeze(clf.predic
t(my_image))]))

y = 1 , sklearn algorithm predicts a cat picture.
```

Sklearn for brest cancer dataset

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
# import load_breast_cancer and get the X_cancer, y_cancer
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
(X_cancer, y_cancer) = cancer.data, cancer.target

#  split to train and test using random_state = 0
X_train, X_test, y_train, y_test = train_test_split(X_cancer,
```

```
y_cancer)
#  train LogisticRegression classifier  for max_iter= 10000
clf = LogisticRegression(max_iter=10000).fit(X_train, y_train)
print('\nBreast cancer dataset')
print('X_cancer.shape= {}'.format(X_cancer.shape))
print('Accuracy of Logistic regression classifier on training set:
{:.2f}'
      .format(clf.score(X_train, y_train)))
print('Accuracy of Logistic regression classifier on test set: {:.2f}'
      .format(clf.score(X_test, y_test)))
```

```
Breast cancer dataset
X_cancer.shape= (569, 30)
Accuracy of Logistic regression classifier on training set: 0.96
Accuracy of Logistic regression classifier on test set: 0.96
```

Expected Output

```
Breast cancer dataset
X_cancer.shape= (569, 30)
Accuracy of Logistic regression classifier on training set: 0.96
Accuracy of Logistic regression classifier on test set: 0.95
```

Sklearn for synthetic dataset

Additional functions for visualization

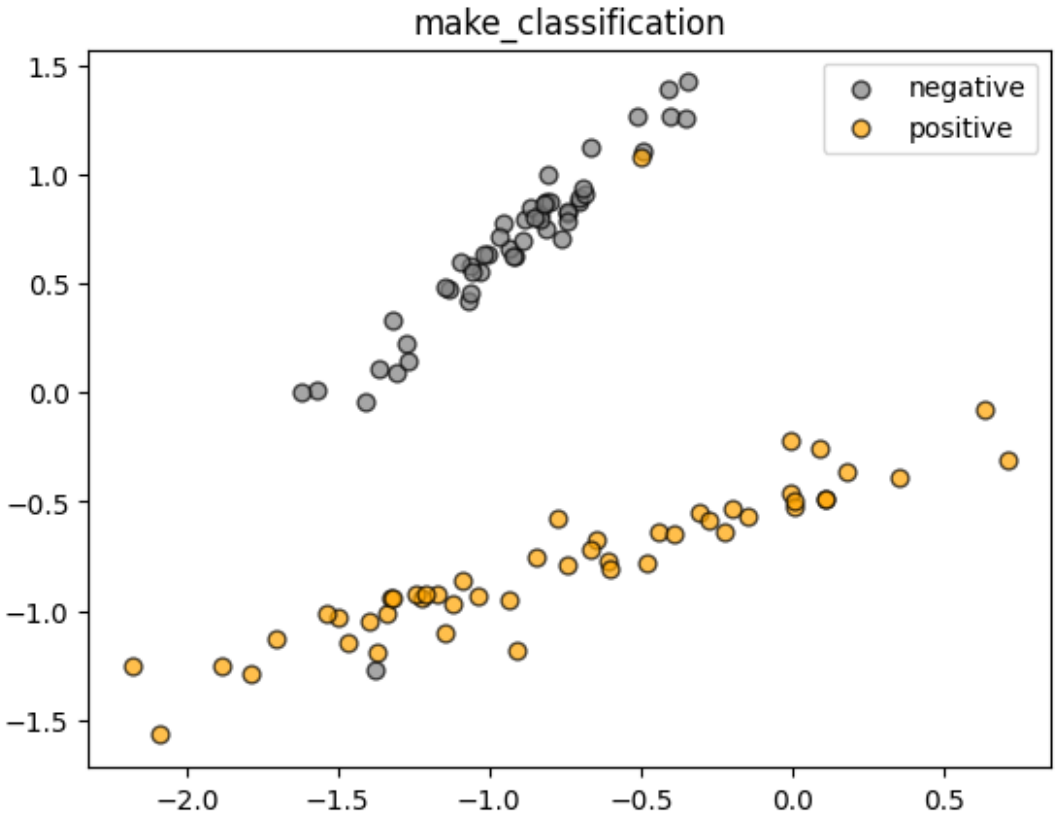
```
%matplotlib notebook

def plot_decision_boundary(clf, X_train, y_train, X_test=None, y_test=
None, title=None, precision=0.01,plot_symbol_size = 50, ax= None,
is_extended=True):

    '''
    Draws the binary decision boundary for X that is nor required
additional features and transformation (like polynomial)
```

```
    '''
    # Create color maps - required by pcolormesh
    from matplotlib.colors import  ListedColormap
    colors_for_points = np.array(['grey', 'orange']) # neg/pos
    colors_for_areas = np.array(['grey', 'orange']) # neg/pos  # alpha
is applied later
    cmap_light = ListedColormap(colors_for_areas)



    mesh_step_size = precision #.01  # step size in the mesh
    if X_test is None or y_test is None:
        show_test= False
        X= X_train
    else:
        show_test= True
        X= np.concatenate([X_train,X_test], axis=0)
    x1_min, x1_max = X[:, 0].min() - .1, X[:, 0].max() + 0.1
    x2_min, x2_max = X[:, 1].min() - .1, X[:, 1].max() + 0.1
    # Create grids of pairs
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, mesh_step_size),
                           np.arange(x2_min, x2_max, mesh_step_size))
    # Flatten all samples
    target_samples_grid= (np.c_[xx1.ravel(), xx2.ravel()])

    print ('Call prediction for all grid values (precision of drawing
= {},\n you may configure to speed up e.g.
precision=0.05)'.format(precision))
    Z = clf.predict(target_samples_grid)

    # Reshape the result to original meshgrid shape
    Z = Z.reshape(xx1.shape)

    if ax:
        plt.sca(ax)

    # Plot all meshgrid prediction
    plt.pcolormesh(xx1, xx2,Z, cmap = cmap_light, alpha=0.2)

    # Plot train set
    plt.scatter(X_train[:, 0], X_train[:, 1], s=plot_symbol_size,
                c=colors_for_points[y_train.ravel()], edgecolor =
'black',alpha=0.6)
    # Plot test set
    if show_test:
        plt.scatter(X_test[:, 0], X_test[:, 1], marker='^',
s=plot_symbol_size,
                c=colors_for_points[y_test.ravel()],edgecolor =
'black',alpha=0.6)
```

```
    if  is_extended:
        # Create legend
        import matplotlib.patches as mpatches # use to assign lavels
for colored points
        patch0 = mpatches.Patch(color=colors_for_points[0],
label='negative')
        patch1 = mpatches.Patch(color=colors_for_points[1],
label='positive')
        plt.legend(handles=[patch0, patch1])
    plt.title(title)
    if is_extended:
        plt.xlabel('feature 1')
        plt.ylabel('feature 2')
    else:
        plt.tick_params(
        top =False,
        bottom= False,
        left  = False,
        labelleft = False,
        labelbottom = False
        )



def plot_data_logistic_regression(X,y,legend_loc= 1, title= None):
    '''
    :param X: 2 dimensional ndarray
    :param y:  1 dimensional ndarray. Use y.ravel() if necessary
    :return:
    '''

    positive_indices = (y == 1)
    negative_indices = (y == 0)
#     import matplotlib as mpl
    colors_for_points = ['grey', 'orange'] # neg/pos

    plt.scatter(X[negative_indices][:,0], X[negative_indices][:,1],
s=40, c=colors_for_points [0], edgecolor = 'black', label='negative',
alpha = 0.7)
    plt.scatter(X[positive_indices][:,0], X[positive_indices][:,1],
s=40, c=colors_for_points [1], edgecolor = 'black',label='positive',
alpha = 0.7)
    plt.title(title)
    plt.legend(loc= legend_loc)
```

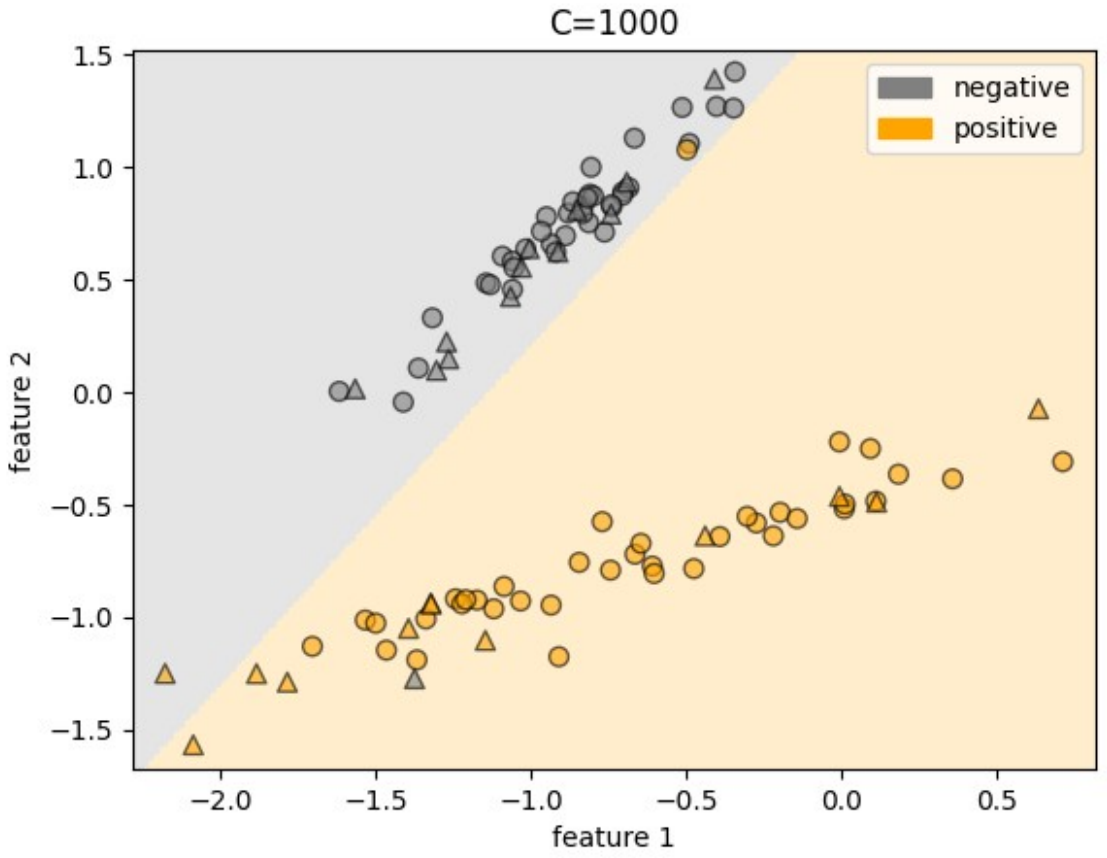
Make classification

```
from sklearn.datasets import make_classification
Xc_2, yc_2= make_classification(n_samples=100,
                                n_features=2,
                                n_informative=2,
                                n_redundant=0,
                                random_state=2018,
                                n_clusters_per_class=1,
                                flip_y = 0.1,
                                class_sep = 0.8)

plt.figure()
plot_data_logistic_regression(Xc_2, yc_2, title=
'make_classification')
```



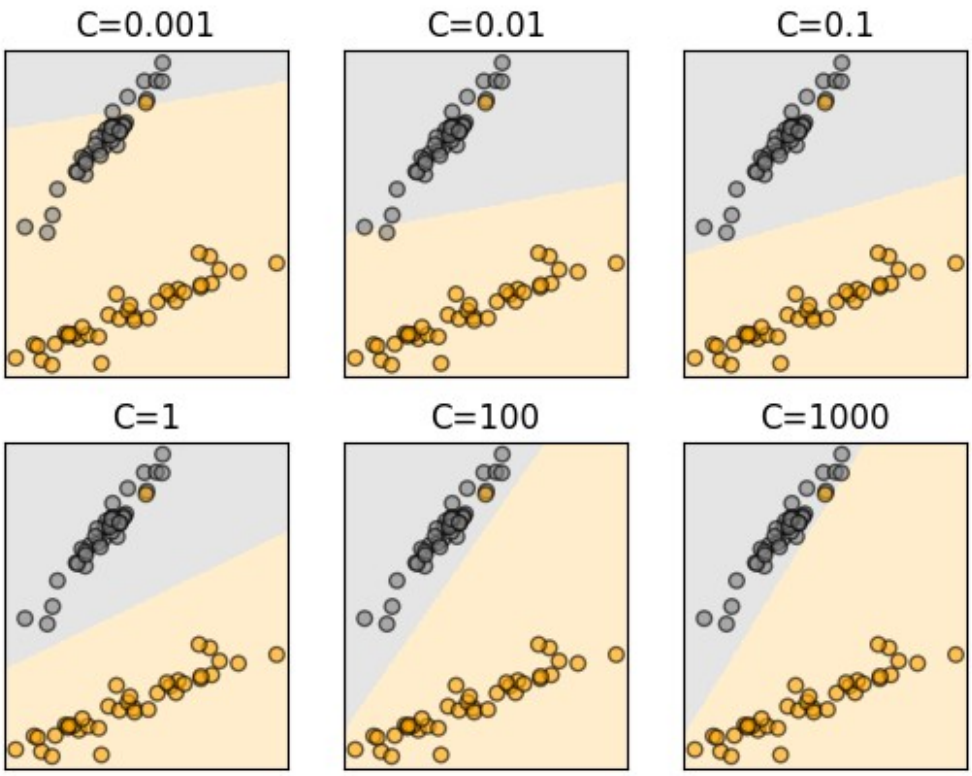

```
C = 1000
X_train, X_test, y_train, y_test = train_test_split(Xc_2, yc_2,
random_state = 0)
clf = LogisticRegression(C=C).fit(X_train, y_train)
print('Make Regression')
print ('Xc_2.shape= {}'.format(Xc_2.shape))
print('Accuracy of Logistic regression classifier on training set:
{:.2f}'
     .format(clf.score(X_train, y_train)))
```

```
print('Accuracy of Logistic regression classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))

Make Regression
Xc_2.shape= (100, 2)
Accuracy of Logistic regression classifier on training set: 0.99
Accuracy of Logistic regression classifier on test set: 0.92

plt.figure()
plot_decision_boundary(clf, X_train, y_train, X_test, y_test,
title='C={}'.format(C),precision=0.01, plot_symbol_size = 50)

Call prediction for all grid values (precision of drawing = 0.01,
 you may configure to speed up e.g. precision=0.05)
```




```
_, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(2, 3)
axes = (ax1, ax2, ax3, ax4, ax5, ax6)
C = (0.001, 0.01, 0.1, 1, 100, 1000)
for i in range (len(C)):
    clf = LogisticRegression(C=C[i]).fit(X_train, y_train)
    print('Accuracy = {:.2f}'.format(clf.score(X_train, y_train)))

    plot_decision_boundary(clf, X_train, y_train,
```

```
title='C={}'.format(C[i]),precision=0.01, plot_symbol_size = 30, ax=
axes[i], is_extended=False)


Accuracy = 0.57
Call prediction for all grid values (precision of drawing = 0.01,
 you may configure to speed up e.g. precision=0.05)
Accuracy = 0.97
Call prediction for all grid values (precision of drawing = 0.01,
 you may configure to speed up e.g. precision=0.05)
Accuracy = 0.99
Call prediction for all grid values (precision of drawing = 0.01,
 you may configure to speed up e.g. precision=0.05)
Accuracy = 0.99
Call prediction for all grid values (precision of drawing = 0.01,
 you may configure to speed up e.g. precision=0.05)
Accuracy = 0.99
Call prediction for all grid values (precision of drawing = 0.01,
 you may configure to speed up e.g. precision=0.05)
Accuracy = 0.99
Call prediction for all grid values (precision of drawing = 0.01,
 you may configure to speed up e.g. precision=0.05)
```



Polynomial In Logistic Regression

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

Additional functions for visualization

```
%matplotlib notebook

def plot_decision_boundary(clf, X_train, y_train, X_test=None, y_test=
None, title=None, precision=0.05,plot_symbol_size = 50, ax= None,
is_extended=True):

    '''
    Draws the binary decision boundary for X that is nor required
additional features and transformation (like polynomial)
    '''
    # Create color maps - required by pcolormesh
    from matplotlib.colors import  ListedColormap
    colors_for_points = np.array(['grey', 'orange']) # neg/pos
    colors_for_areas = np.array(['grey', 'orange']) # neg/pos  # alpha
is applied later
    cmap_light = ListedColormap(colors_for_areas)



    mesh_step_size = precision #.01  # step size in the mesh
    if X_test is None or y_test is None:
        show_test= False
        X= X_train
    else:
        show_test= True
        X= np.concatenate([X_train,X_test], axis=0)
    x1_min, x1_max = X[:, 0].min() - .1, X[:, 0].max() + 0.1
    x2_min, x2_max = X[:, 1].min() - .1, X[:, 1].max() + 0.1
    # Create grids of pairs
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, mesh_step_size),
                           np.arange(x2_min, x2_max, mesh_step_size))
    # Flatten all samples
    target_samples_grid= (np.c_[xx1.ravel(), xx2.ravel()])
```

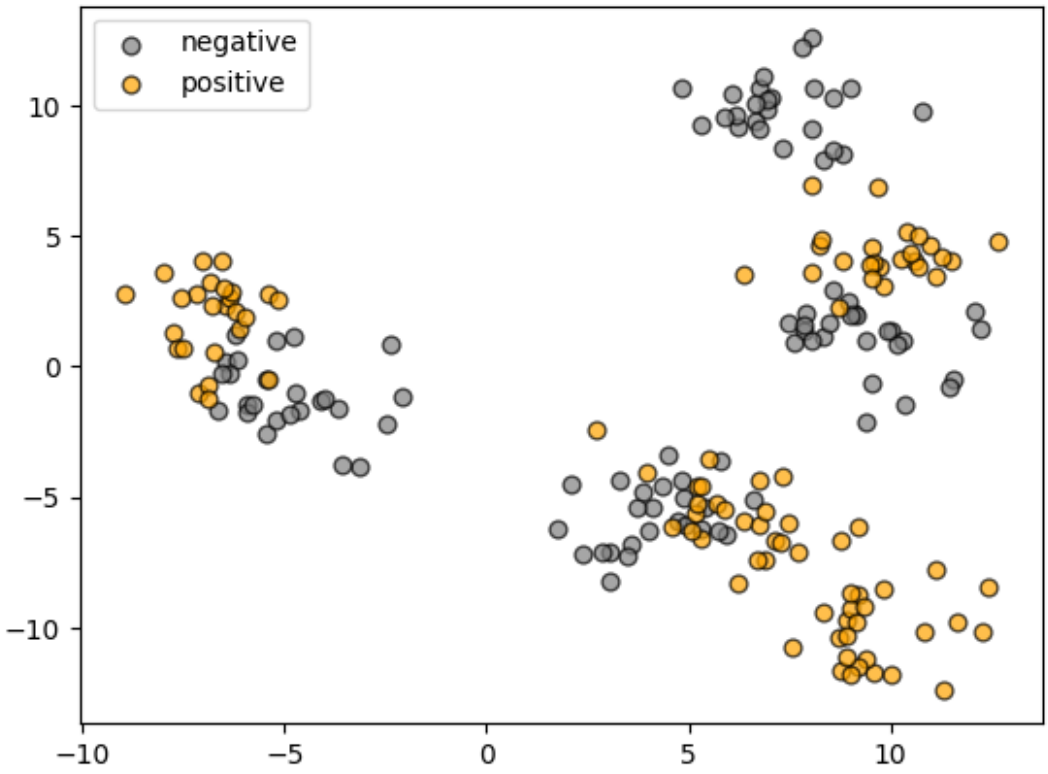
```
    print ('Call prediction for all grid values (precision of drawing
= {},\n you may configure to speed up e.g.
precision=0.05)'.format(precision))
    Z = clf.predict(target_samples_grid)

    # Reshape the result to original meshgrid shape
    Z = Z.reshape(xx1.shape)

    if ax:
        plt.sca(ax)

    # Plot all meshgrid prediction
    plt.pcolormesh(xx1, xx2,Z, cmap = cmap_light, alpha=0.2)

    # Plot train set
    plt.scatter(X_train[:, 0], X_train[:, 1], s=plot_symbol_size,
                c=colors_for_points[y_train.ravel()], edgecolor =
'black',alpha=0.6)
    # Plot test set
    if show_test:
        plt.scatter(X_test[:, 0], X_test[:, 1], marker='^',
s=plot_symbol_size,
                c=colors_for_points[y_test.ravel()],edgecolor =
'black',alpha=0.6)
    if  is_extended:
        # Create legend
        import matplotlib.patches as mpatches # use to assign lavels
for colored points
        patch0 = mpatches.Patch(color=colors_for_points[0],
label='negative')
        patch1 = mpatches.Patch(color=colors_for_points[1],
label='positive')
        plt.legend(handles=[patch0, patch1])
    plt.title(title)
    if is_extended:
        plt.xlabel('feature 1')
        plt.ylabel('feature 2')
    else:
        plt.tick_params(
        top =False,
        bottom= False,
        left  = False,
        labelleft = False,
        labelbottom = False
        )

def plot_decision_boundary_poly(clf, X_train, y_train, degree,
X_test=None, y_test= None, title=None, precision=0.05,plot_symbol_size
= 50, ax= None,  is_extended=True):
```

```
    '''
    Draws the binary decision boundary for X that is nor required
additional features and transformation (like polynomial)
    '''
    from sklearn.preprocessing import PolynomialFeatures
    poly = PolynomialFeatures(degree=degree,include_bias=False)

    # Create color maps - required by pcolormesh
    from matplotlib.colors import  ListedColormap
    colors_for_points = np.array(['grey', 'orange']) # neg/pos
    colors_for_areas = np.array(['grey', 'orange']) # neg/pos  # alpha
is applied later
    cmap_light = ListedColormap(colors_for_areas)



    mesh_step_size = precision #.01  # step size in the mesh
    if X_test is None or y_test is None:
        show_test= False
        X= X_train
    else:
        show_test= True
        X= np.concatenate([X_train,X_test], axis=0)
    x1_min, x1_max = X[:, 0].min() - .1, X[:, 0].max() + 0.1
    x2_min, x2_max = X[:, 1].min() - .1, X[:, 1].max() + 0.1
    # Create grids of pairs
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, mesh_step_size),
                           np.arange(x2_min, x2_max, mesh_step_size))
    # Flatten all samples
    target_samples_grid= (np.c_[xx1.ravel(), xx2.ravel()])
    target_samples_grid_poly = poly.fit_transform(target_samples_grid)
    print ('Call prediction for all grid values (precision of drawing
= {},\n you may configure to speed up e.g.
precision=0.05)'.format(precision))
    Z = clf.predict(target_samples_grid_poly)
    print ('Computing prediction completed.')
    # Reshape the result to original meshgrid shape
    Z = Z.reshape(xx1.shape)

    if ax:
        plt.sca(ax)

    # Plot all meshgrid prediction
    plt.pcolormesh(xx1, xx2,Z, cmap = cmap_light, alpha=0.2)

    # Plot train set
    plt.scatter(X_train[:, 0], X_train[:, 1], s=plot_symbol_size,
                c=colors_for_points[y_train.ravel()], edgecolor =
```

```
'black',alpha=0.6)
    # Plot test set
    if show_test:
        plt.scatter(X_test[:, 0], X_test[:, 1], marker='^',
s=plot_symbol_size,
                    c=colors_for_points[y_test.ravel()],edgecolor =
'black',alpha=0.6)
    if  is_extended:
        # Create legend
        import matplotlib.patches as mpatches # use to assign lavels
for colored points
        patch0 = mpatches.Patch(color=colors_for_points[0],
label='negative')
        patch1 = mpatches.Patch(color=colors_for_points[1],
label='positive')
        plt.legend(handles=[patch0, patch1])
    plt.title(title)
    if is_extended:
        plt.xlabel('feature 1')
        plt.ylabel('feature 2')
    else:
        plt.tick_params(
        top =False,
        bottom= False,
        left  = False,
        labelleft = False,
        labelbottom = False
        )



def plot_data_logistic_regression(X,y,legend_loc= None, title= None):
    '''
    :param X: 2 dimensional ndarray
    :param y:  1 dimensional ndarray. Use y.ravel() if necessary
    :return:
    '''

    positive_indices = (y == 1)
    negative_indices = (y == 0)
#     import matplotlib as mpl
    colors_for_points = ['grey', 'orange'] # neg/pos

    plt.scatter(X[negative_indices][:,0], X[negative_indices][:,1],
s=40, c=colors_for_points [0], edgecolor = 'black', label='negative',
alpha = 0.7)
    plt.scatter(X[positive_indices][:,0], X[positive_indices][:,1],
s=40, c=colors_for_points [1], edgecolor = 'black',label='positive',
alpha = 0.7)
```

```
    plt.title(title)
    plt.legend(loc= legend_loc)

def plot_multi_class_logistic_regression(X,y,dict_names=None, colors=
None,  title =None):
    '''
    Draw the multi class samples of 2 features
    :param X: X 2 ndarray (m,2),
    :param y: vector (m,)
    :param dict_names: dict of values of y and names
    :return: None
    '''
    if not colors:
        colors_for_points = ['green','grey', 'orange', 'brown']
    else:
         colors_for_points = colors

    y_unique = list(set(y))

    for i in range (len(y_unique)):
        ind = y == y_unique[i] # vector

        if dict_names:
            plt.scatter(X[ind,0], X[ind,1], c=colors_for_points[i],
s=40, label=dict_names[y_unique[i]],edgecolor='black', alpha=.7)
        else:
            plt.scatter(X[ind, 0], X[ind, 1], s=40,
c=colors_for_points [i], edgecolor = 'black', alpha = 0.7)
    if title:
        plt.title(title)

    if dict_names:
        plt.legend(frameon=True)

def draw_linear_decision_boundaries_multiclass(clf,X,):
    colors=  ['green','grey', 'orange', 'brown']
    x_line = np.linspace(X[:,0].min(),X[:,0].max(), 100)
    for w, b, color in zip(clf.coef_, clf.intercept_, colors):
        # Since class prediction with a linear model uses the formula
y = w_0 x_0 + w_1 x_1 + b,
        # and the decision boundary is defined as being all points
with y = 0, to plot x_1 as a
        # function of x_0 we just solve w_0 x_0 + w_1 x_1 + b = 0 for
x_1:
        y_line = -(x_line  * w[0] + b) / w[1]
        ind = (X[:,0].min()< x_line) & (x_line <X[:,0].max()) &
(X[:,1].min()< y_line) & (y_line <X[:,1].max() )
        plt.plot(x_line[ind] , y_line[ind], '-', c=color, alpha=.8)

from sklearn.datasets import make_blobs
```

```
X_mk8, y_mk8 = make_blobs(n_samples = 200, n_features = 2, centers =
8, # centers impacts for y
                          cluster_std = 1.3, random_state = 4)
y_train = y_mk8 % 2 # make it binary since make_blobs  with centers =
8 creates y in [0..7]

plt.figure()
plot_data_logistic_regression(X_mk8,y_train)
```



Polynomial Features

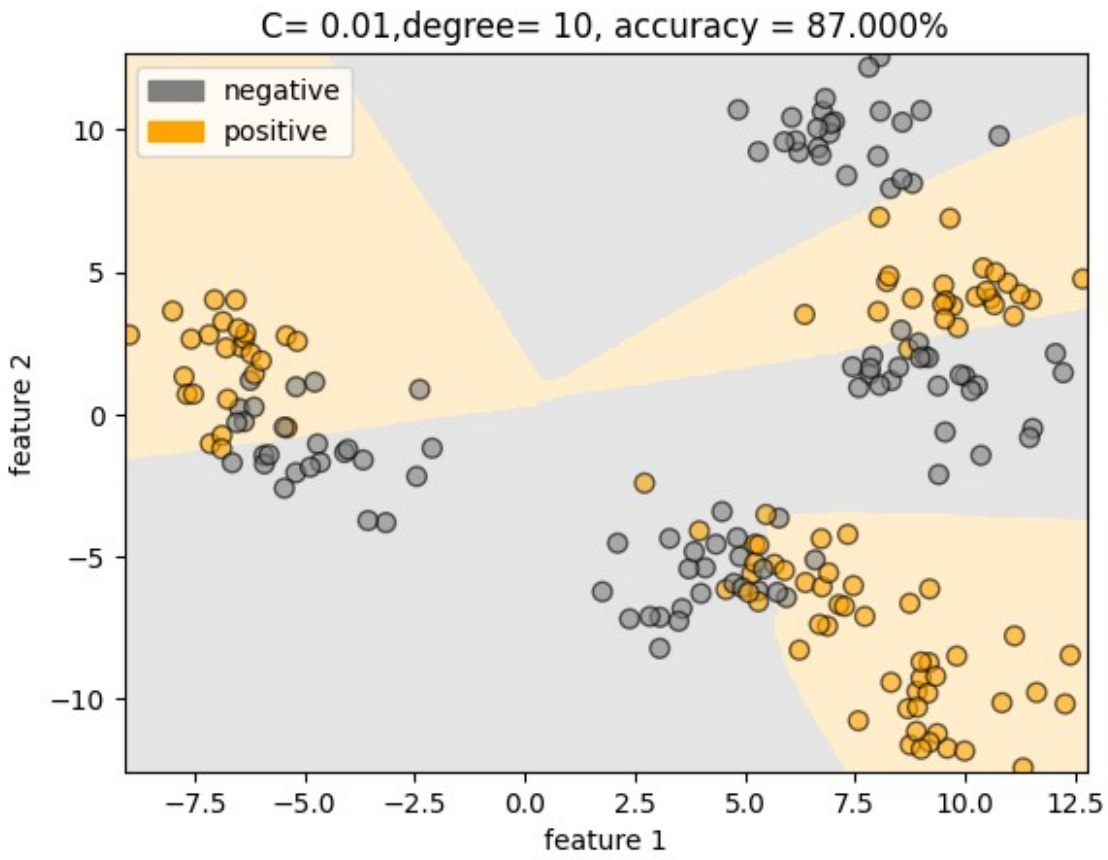
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LogisticRegression

degree = 10
poly= PolynomialFeatures(degree=degree,include_bias=False) # default
is True means to return the first feature of all 1 as for degree 0
X_train_poly= poly.fit_transform(X_mk8)

C = 0.01
clf = LogisticRegression(C=C).fit(X_train_poly, y_train)
```

```
accuracy = clf.score (X_train_poly, y_train)
# print("train accuracy= {:.3%}".format(accuracy))
```

```
c:\Users\Smeek\AppData\Local\Programs\Python\Python312\Lib\site-
packages\sklearn\linear_model\_logistic.py:460: ConvergenceWarning:
lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as
shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

```
plt.figure()
ax = plt.gca()
plot_decision_boundary_poly(clf, X_train_poly, y_train, degree=
degree, ax = ax, precision= 0.05, title = 'C= {},degree= {}, accuracy
= {:.3%}'.format(C, degree, accuracy))
```

```
Call prediction for all grid values (precision of drawing = 0.05,
 you may configure to speed up e.g. precision=0.05)
Computing prediction completed.
```

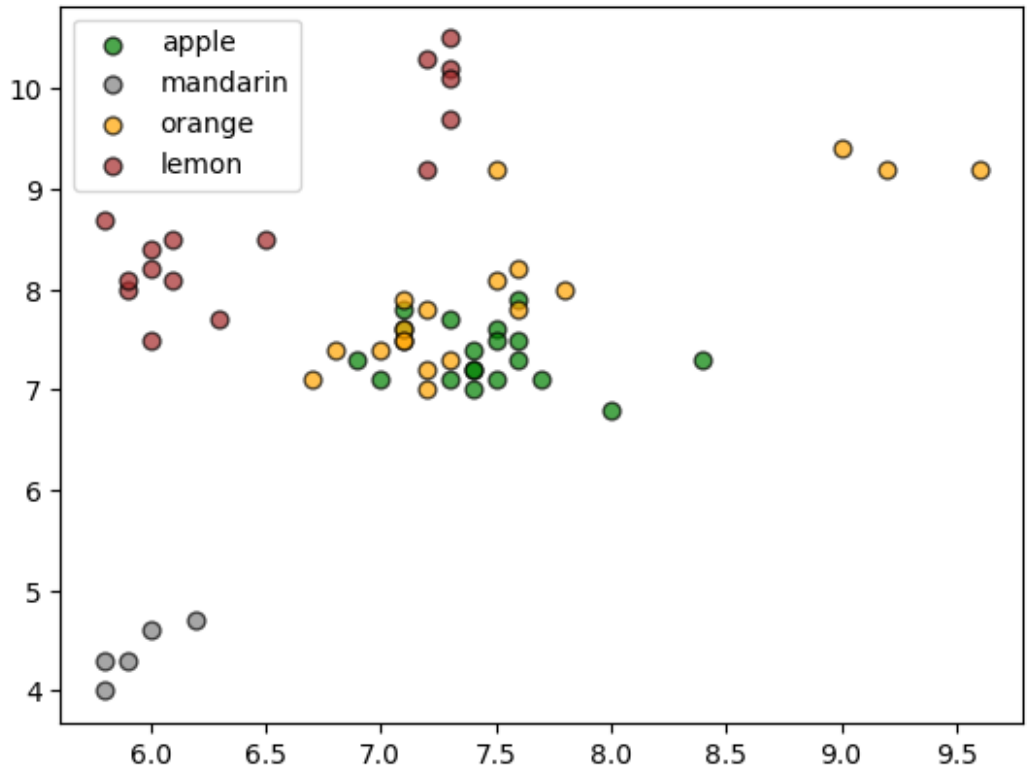


Muticlass classification

Load fruits data set

```
import os
cwd= os.getcwd() # current working directory
path = os.path.join(cwd,'data')
fn=  os.path.join(path , 'fruit_data_with_colors.txt')
df_fruits = pd.read_table(fn)
X = df_fruits[['width', 'height']].values
y = df_fruits['fruit_label'].values
df_fruits.head(20)
fruits_dict = dict(zip(df_fruits['fruit_label'].unique(),
df_fruits['fruit_name'].unique()))
# fruits_dict
```

```
plt.figure()
plot_multi_class_logistic_regression (X,y,dict_names=fruits_dict)
```




```
help(LogisticRegression)

Help on class LogisticRegression in module
sklearn.linear_model._logistic:

class
LogisticRegression(sklearn.linear_model._base.LinearClassifierMixin,
sklearn.linear_model._base.SparseCoefMixin,
sklearn.base.BaseEstimator)
 |  LogisticRegression(penalty='l2', *, dual=False, tol=0.0001, C=1.0,
fit_intercept=True, intercept_scaling=1, class_weight=None,
random_state=None, solver='lbfgs', max_iter=100, multi_class='auto',
verbose=0, warm_start=False, n_jobs=None, l1_ratio=None)
 |
 |  Logistic Regression (aka logit, MaxEnt) classifier.
 |
 |  In the multiclass case, the training algorithm uses the one-vs-
rest (OvR)
 |  scheme if the 'multi_class' option is set to 'ovr', and uses the
 |  cross-entropy loss if the 'multi_class' option is set to
'multinomial'.
 |  (Currently the 'multinomial' option is supported only by the
```

```
'lbfgs',
 |  'sag', 'saga' and 'newton-cg' solvers.)
 |
 |  This class implements regularized logistic regression using the
 |  'liblinear' library, 'newton-cg', 'sag', 'saga' and 'lbfgs'
solvers. **Note
 |  that regularization is applied by default**. It can handle both
dense
 |  and sparse input. Use C-ordered arrays or CSR matrices containing
64-bit
 |  floats for optimal performance; any other input format will be
converted
 |  (and copied).
 |
 |  The 'newton-cg', 'sag', and 'lbfgs' solvers support only L2
regularization
 |  with primal formulation, or no regularization. The 'liblinear'
solver
 |  supports both L1 and L2 regularization, with a dual formulation
only for
 |  the L2 penalty. The Elastic-Net regularization is only supported
by the
 |  'saga' solver.
 |
 |  Read more in the :ref:`User Guide <logistic_regression>`.
 |
 |  Parameters
 |  ----------
 |  penalty : {'l1', 'l2', 'elasticnet', None}, default='l2'
 |      Specify the norm of the penalty:
 |
 |      - `None`: no penalty is added;
 |      - `'l2'`: add a L2 penalty term and it is the default choice;
 |      - `'l1'`: add a L1 penalty term;
 |      - `'elasticnet'`: both L1 and L2 penalty terms are added.
 |
 |      .. warning::
 |         Some penalties may not work with some solvers. See the
parameter
 |         `solver` below, to know the compatibility between the
penalty and
 |         solver.
 |
 |      .. versionadded:: 0.19
 |         l1 penalty with SAGA solver (allowing 'multinomial' + L1)
 |
 |      .. deprecated:: 1.2
 |         The 'none' option was deprecated in version 1.2, and will
be removed
```

```
 |         in 1.4. Use `None` instead.
 |
 |  dual : bool, default=False
 |      Dual (constrained) or primal (regularized, see also
 |      :ref:`this equation <regularized-logistic-loss>`) formulation.
Dual formulation
 |      is only implemented for l2 penalty with liblinear solver.
Prefer dual=False when
 |      n_samples > n_features.
 |
 |  tol : float, default=1e-4
 |      Tolerance for stopping criteria.
 |
 |  C : float, default=1.0
 |      Inverse of regularization strength; must be a positive float.
 |      Like in support vector machines, smaller values specify
stronger
 |      regularization.
 |
 |  fit_intercept : bool, default=True
 |      Specifies if a constant (a.k.a. bias or intercept) should be
 |      added to the decision function.
 |
 |  intercept_scaling : float, default=1
 |      Useful only when the solver 'liblinear' is used
 |      and self.fit_intercept is set to True. In this case, x becomes
 |      [x, self.intercept_scaling],
 |      i.e. a "synthetic" feature with constant value equal to
 |      intercept_scaling is appended to the instance vector.
 |      The intercept becomes ``intercept_scaling *
synthetic_feature_weight``.
 |
 |      Note! the synthetic feature weight is subject to l1/l2
regularization
 |      as all other features.
 |      To lessen the effect of regularization on synthetic feature
weight
 |      (and therefore on the intercept) intercept_scaling has to be
increased.
 |
 |  class_weight : dict or 'balanced', default=None
 |      Weights associated with classes in the form ``{class_label:
weight}``.
 |      If not given, all classes are supposed to have weight one.
 |
 |      The "balanced" mode uses the values of y to automatically
adjust
 |      weights inversely proportional to class frequencies in the
input data
```

```
 |      as ``n_samples / (n_classes * np.bincount(y))``.
 |
 |      Note that these weights will be multiplied with sample_weight
(passed
 |      through the fit method) if sample_weight is specified.
 |
 |      .. versionadded:: 0.17
 |         *class_weight='balanced'*
 |
 |  random_state : int, RandomState instance, default=None
 |      Used when ``solver`` == 'sag', 'saga' or 'liblinear' to
shuffle the
 |      data. See :term:`Glossary <random_state>` for details.
 |
 |  solver : {'lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky',
'sag', 'saga'},             default='lbfgs'
 |
 |      Algorithm to use in the optimization problem. Default is
'lbfgs'.
 |      To choose a solver, you might want to consider the following
aspects:
 |
 |          - For small datasets, 'liblinear' is a good choice,
whereas 'sag'
 |            and 'saga' are faster for large ones;
 |          - For multiclass problems, only 'newton-cg', 'sag', 'saga'
and
 |            'lbfgs' handle multinomial loss;
 |          - 'liblinear' is limited to one-versus-rest schemes.
 |          - 'newton-cholesky' is a good choice for `n_samples` >>
`n_features`,
 |            especially with one-hot encoded categorical features
with rare
 |            categories. Note that it is limited to binary
classification and the
 |            one-versus-rest reduction for multiclass classification.
Be aware that
 |            the memory usage of this solver has a quadratic
dependency on
 |            `n_features` because it explicitly computes the Hessian
matrix.
 |
 |      .. warning::
 |         The choice of the algorithm depends on the penalty chosen.
 |         Supported penalties by solver:
 |
 |         - 'lbfgs'           -   ['l2', None]
 |         - 'liblinear'       -   ['l1', 'l2']
 |         - 'newton-cg'       -   ['l2', None]
```

```
 |         - 'newton-cholesky' -   ['l2', None]
 |         - 'sag'             -   ['l2', None]
 |         - 'saga'            -   ['elasticnet', 'l1', 'l2', None]
 |
 |      .. note::
 |         'sag' and 'saga' fast convergence is only guaranteed on
features
 |         with approximately the same scale. You can preprocess the
data with
 |         a scaler from :mod:`sklearn.preprocessing`.
 |
 |      .. seealso::
 |         Refer to the User Guide for more information regarding
 |         :class:`LogisticRegression` and more specifically the
 |         :ref:`Table <Logistic_regression>`
 |         summarizing solver/penalty supports.
 |
 |      .. versionadded:: 0.17
 |         Stochastic Average Gradient descent solver.
 |      .. versionadded:: 0.19
 |         SAGA solver.
 |      .. versionchanged:: 0.22
 |          The default solver changed from 'liblinear' to 'lbfgs' in
0.22.
 |      .. versionadded:: 1.2
 |         newton-cholesky solver.
 |
 |  max_iter : int, default=100
 |      Maximum number of iterations taken for the solvers to
converge.
 |
 |  multi_class : {'auto', 'ovr', 'multinomial'}, default='auto'
 |      If the option chosen is 'ovr', then a binary problem is fit
for each
 |      label. For 'multinomial' the loss minimised is the multinomial
loss fit
 |      across the entire probability distribution, *even when the
data is
 |      binary*. 'multinomial' is unavailable when solver='liblinear'.
 |      'auto' selects 'ovr' if the data is binary, or if
solver='liblinear',
 |      and otherwise selects 'multinomial'.
 |
 |      .. versionadded:: 0.18
 |         Stochastic Average Gradient descent solver for
'multinomial' case.
 |      .. versionchanged:: 0.22
 |          Default changed from 'ovr' to 'auto' in 0.22.
 |
```

```
 |  verbose : int, default=0
 |      For the liblinear and lbfgs solvers set verbose to any
positive
 |      number for verbosity.
 |
 |  warm_start : bool, default=False
 |      When set to True, reuse the solution of the previous call to
fit as
 |      initialization, otherwise, just erase the previous solution.
 |      Useless for liblinear solver. See :term:`the Glossary
<warm_start>`.
 |
 |      .. versionadded:: 0.17
 |         *warm_start* to support *lbfgs*, *newton-cg*, *sag*, *saga*
solvers.
 |
 |  n_jobs : int, default=None
 |      Number of CPU cores used when parallelizing over classes if
 |      multi_class='ovr'". This parameter is ignored when the
``solver`` is
 |      set to 'liblinear' regardless of whether 'multi_class' is
specified or
 |      not. ``None`` means 1 unless in
a :obj:`joblib.parallel_backend`
 |      context. ``-1`` means using all processors.
 |      See :term:`Glossary <n_jobs>` for more details.
 |
 |  l1_ratio : float, default=None
 |      The Elastic-Net mixing parameter, with ``0 <= l1_ratio <= 1``.
Only
 |      used if ``penalty='elasticnet'``. Setting ``l1_ratio=0`` is
equivalent
 |      to using ``penalty='l2'``, while setting ``l1_ratio=1`` is
equivalent
 |      to using ``penalty='l1'``. For ``0 < l1_ratio <1``, the
penalty is a
 |      combination of L1 and L2.
 |
 |  Attributes
 |  ----------
 |
 |  classes_ : ndarray of shape (n_classes, )
 |      A list of class labels known to the classifier.
 |
 |  coef_ : ndarray of shape (1, n_features) or (n_classes,
n_features)
 |      Coefficient of the features in the decision function.
 |
 |      `coef_` is of shape (1, n_features) when the given problem is
```

```
binary.
 |      In particular, when `multi_class='multinomial'`, `coef_`
corresponds
 |      to outcome 1 (True) and `-coef_` corresponds to outcome 0
(False).
 |
 |  intercept_ : ndarray of shape (1,) or (n_classes,)
 |      Intercept (a.k.a. bias) added to the decision function.
 |
 |      If `fit_intercept` is set to False, the intercept is set to
zero.
 |      `intercept_` is of shape (1,) when the given problem is
binary.
 |      In particular, when `multi_class='multinomial'`, `intercept_`
 |      corresponds to outcome 1 (True) and `-intercept_` corresponds
to
 |      outcome 0 (False).
 |
 |  n_features_in_ : int
 |      Number of features seen during :term:`fit`.
 |
 |      .. versionadded:: 0.24
 |
 |  feature_names_in_ : ndarray of shape (`n_features_in_`,)
 |      Names of features seen during :term:`fit`. Defined only when
`X`
 |      has feature names that are all strings.
 |
 |      .. versionadded:: 1.0
 |
 |  n_iter_ : ndarray of shape (n_classes,) or (1, )
 |      Actual number of iterations for all classes. If binary or
multinomial,
 |      it returns only 1 element. For liblinear solver, only the
maximum
 |      number of iteration across all classes is given.
 |
 |      .. versionchanged:: 0.20
 |
 |          In SciPy <= 1.0.0 the number of lbfgs iterations may
exceed
 |          ``max_iter``. ``n_iter_`` will now report at most
``max_iter``.
 |
 |  See Also
 |  --------
 |  SGDClassifier : Incrementally trained logistic regression (when
given
 |      the parameter ``loss="log_loss"``).
```

```
 |  LogisticRegressionCV : Logistic regression with built-in cross
validation.
 |
 |  Notes
 |  -----
 |  The underlying C implementation uses a random number generator to
 |  select features when fitting the model. It is thus not uncommon,
 |  to have slightly different results for the same input data. If
 |  that happens, try with a smaller tol parameter.
 |
 |  Predict output may not match that of standalone liblinear in
certain
 |  cases. See :ref:`differences from liblinear
<liblinear_differences>`
 |  in the narrative documentation.
 |
 |  References
 |  ----------
 |
 |  L-BFGS-B -- Software for Large-scale Bound-constrained
Optimization
 |      Ciyou Zhu, Richard Byrd, Jorge Nocedal and Jose Luis Morales.
 |      http://users.iems.northwestern.edu/~nocedal/lbfgsb.html
 |
 |  LIBLINEAR -- A Library for Large Linear Classification
 |      https://www.csie.ntu.edu.tw/~cjlin/liblinear/
 |
 |  SAG -- Mark Schmidt, Nicolas Le Roux, and Francis Bach
 |      Minimizing Finite Sums with the Stochastic Average Gradient
 |      https://hal.inria.fr/hal-00860051/document
 |
 |  SAGA -- Defazio, A., Bach F. & Lacoste-Julien S. (2014).
 |          :arxiv:`"SAGA: A Fast Incremental Gradient Method With
Support
 |          for Non-Strongly Convex Composite Objectives" <1407.0202>`
 |
 |  Hsiang-Fu Yu, Fang-Lan Huang, Chih-Jen Lin (2011). Dual coordinate
descent
 |      methods for logistic regression and maximum entropy models.
 |      Machine Learning 85(1-2):41-75.
 |      https://www.csie.ntu.edu.tw/~cjlin/papers/maxent_dual.pdf
 |
 |  Examples
 |  --------
 |  >>> from sklearn.datasets import load_iris
 |  >>> from sklearn.linear_model import LogisticRegression
 |  >>> X, y = load_iris(return_X_y=True)
 |  >>> clf = LogisticRegression(random_state=0).fit(X, y)
 |  >>> clf.predict(X[:2, :])
```

```
 |  array([0, 0])
 |  >>> clf.predict_proba(X[:2, :])
 |  array([[9.8...e-01, 1.8...e-02, 1.4...e-08],
 |         [9.7...e-01, 2.8...e-02, ...e-08]])
 |  >>> clf.score(X, y)
 |  0.97...
 |
 |  Method resolution order:
 |      LogisticRegression
 |      sklearn.linear_model._base.LinearClassifierMixin
 |      sklearn.base.ClassifierMixin
 |      sklearn.linear_model._base.SparseCoefMixin
 |      sklearn.base.BaseEstimator
 |      sklearn.utils._metadata_requests._MetadataRequester
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, penalty='l2', *, dual=False, tol=0.0001, C=1.0,
fit_intercept=True, intercept_scaling=1, class_weight=None,
random_state=None, solver='lbfgs', max_iter=100, multi_class='auto',
verbose=0, warm_start=False, n_jobs=None, l1_ratio=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  fit(self, X, y, sample_weight=None)
 |      Fit the model according to the given training data.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples,
n_features)
 |          Training vector, where `n_samples` is the number of
samples and
 |          `n_features` is the number of features.
 |
 |      y : array-like of shape (n_samples,)
 |          Target vector relative to X.
 |
 |      sample_weight : array-like of shape (n_samples,) default=None
 |          Array of weights that are assigned to individual samples.
 |          If not provided, then each sample is given unit weight.
 |
 |          .. versionadded:: 0.17
 |             *sample_weight* support to LogisticRegression.
 |
 |      Returns
 |      -------
 |      self
 |          Fitted estimator.
```

```
 |
 |      Notes
 |      -----
 |      The SAGA solver supports both float64 and float32 bit arrays.
 |
 |  predict_log_proba(self, X)
 |      Predict logarithm of probability estimates.
 |
 |      The returned estimates for all classes are ordered by the
 |      label of classes.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Vector to be scored, where `n_samples` is the number of
samples and
 |          `n_features` is the number of features.
 |
 |      Returns
 |      -------
 |      T : array-like of shape (n_samples, n_classes)
 |          Returns the log-probability of the sample for each class
in the
 |          model, where classes are ordered as they are in
``self.classes_``.
 |
 |  predict_proba(self, X)
 |      Probability estimates.
 |
 |      The returned estimates for all classes are ordered by the
 |      label of classes.
 |
 |      For a multi_class problem, if multi_class is set to be
"multinomial"
 |      the softmax function is used to find the predicted probability
of
 |      each class.
 |      Else use a one-vs-rest approach, i.e calculate the probability
 |      of each class assuming it to be positive using the logistic
function.
 |      and normalize these values across all the classes.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Vector to be scored, where `n_samples` is the number of
samples and
 |          `n_features` is the number of features.
 |
```

```
 |      Returns
 |      -------
 |      T : array-like of shape (n_samples, n_classes)
 |          Returns the probability of the sample for each class in
the model,
 |          where classes are ordered as they are in
``self.classes_``.
 |
 |  set_fit_request(self:
sklearn.linear_model._logistic.LogisticRegression, *, sample_weight:
Union[bool, NoneType, str] = '$UNCHANGED$') ->
sklearn.linear_model._logistic.LogisticRegression
 |      Request metadata passed to the ``fit`` method.
 |
 |      Note that this method is only relevant if
 |      ``enable_metadata_routing=True``
(see :func:`sklearn.set_config`).
 |      Please see :ref:`User Guide <metadata_routing>` on how the
routing
 |      mechanism works.
 |
 |      The options for each parameter are:
 |
 |      - ``True``: metadata is requested, and passed to ``fit`` if
provided. The request is ignored if metadata is not provided.
 |
 |      - ``False``: metadata is not requested and the meta-estimator
will not pass it to ``fit``.
 |
 |      - ``None``: metadata is not requested, and the meta-estimator
will raise an error if the user provides it.
 |
 |      - ``str``: metadata should be passed to the meta-estimator
with this given alias instead of the original name.
 |
 |      The default (``sklearn.utils.metadata_routing.UNCHANGED``)
retains the
 |      existing request. This allows you to change the request for
some
 |      parameters and not others.
 |
 |      .. versionadded:: 1.3
 |
 |      .. note::
 |          This method is only relevant if this estimator is used as
a
 |          sub-estimator of a meta-estimator, e.g. used inside a
 |          :class:`~sklearn.pipeline.Pipeline`. Otherwise it has no
effect.
 |
```

```
 |      Parameters
 |      ----------
 |      sample_weight : str, True, False, or None,
default=sklearn.utils.metadata_routing.UNCHANGED
 |          Metadata routing for ``sample_weight`` parameter in
``fit``.
 |
 |      Returns
 |      -------
 |      self : object
 |          The updated object.
 |
 |  set_score_request(self:
sklearn.linear_model._logistic.LogisticRegression, *, sample_weight:
Union[bool, NoneType, str] = '$UNCHANGED$') ->
sklearn.linear_model._logistic.LogisticRegression
 |      Request metadata passed to the ``score`` method.
 |
 |      Note that this method is only relevant if
 |      ``enable_metadata_routing=True``
(see :func:`sklearn.set_config`).
 |      Please see :ref:`User Guide <metadata_routing>` on how the
routing
 |      mechanism works.
 |
 |      The options for each parameter are:
 |
 |      - ``True``: metadata is requested, and passed to ``score`` if
provided. The request is ignored if metadata is not provided.
 |
 |      - ``False``: metadata is not requested and the meta-estimator
will not pass it to ``score``.
 |
 |      - ``None``: metadata is not requested, and the meta-estimator
will raise an error if the user provides it.
 |
 |      - ``str``: metadata should be passed to the meta-estimator
with this given alias instead of the original name.
 |
 |      The default (``sklearn.utils.metadata_routing.UNCHANGED``)
retains the
 |      existing request. This allows you to change the request for
some
 |      parameters and not others.
 |
 |      .. versionadded:: 1.3
 |
 |      .. note::
 |          This method is only relevant if this estimator is used as
```

```
a
 |          sub-estimator of a meta-estimator, e.g. used inside a
 |          :class:`~sklearn.pipeline.Pipeline`. Otherwise it has no
effect.
 |
 |      Parameters
 |      ----------
 |      sample_weight : str, True, False, or None,
default=sklearn.utils.metadata_routing.UNCHANGED
 |          Metadata routing for ``sample_weight`` parameter in
``score``.
 |
 |      Returns
 |      -------
 |      self : object
 |          The updated object.
 |
 |
----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __annotations__ = {'_parameter_constraints': <class 'dict'>}
 |
 |
----------------------------------------------------------------------
 |  Methods inherited from
sklearn.linear_model._base.LinearClassifierMixin:
 |
 |  decision_function(self, X)
 |      Predict confidence scores for samples.
 |
 |      The confidence score for a sample is proportional to the
signed
 |      distance of that sample to the hyperplane.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples,
n_features)
 |          The data matrix for which we want to get the confidence
scores.
 |
 |      Returns
 |      -------
 |      scores : ndarray of shape (n_samples,) or (n_samples,
n_classes)
 |          Confidence scores per `(n_samples, n_classes)`
combination. In the
 |          binary case, confidence score for `self.classes_[1]` where
```

```
>0 means
 |          this class would be predicted.
 |
 |  predict(self, X)
 |      Predict class labels for samples in X.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples,
n_features)
 |          The data matrix for which we want to get the predictions.
 |
 |      Returns
 |      -------
 |      y_pred : ndarray of shape (n_samples,)
 |          Vector containing the class labels for each sample.
 |
 |
----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.ClassifierMixin:
 |
 |  score(self, X, y, sample_weight=None)
 |      Return the mean accuracy on the given test data and labels.
 |
 |      In multi-label classification, this is the subset accuracy
 |      which is a harsh metric since you require for each sample that
 |      each label set be correctly predicted.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Test samples.
 |
 |      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
 |          True labels for `X`.
 |
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Sample weights.
 |
 |      Returns
 |      -------
 |      score : float
 |          Mean accuracy of ``self.predict(X)`` w.r.t. `y`.
 |
 |
----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.ClassifierMixin:
 |
 |  __dict__
```

```
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |
----------------------------------------------------------------------
 |  Methods inherited from sklearn.linear_model._base.SparseCoefMixin:
 |
 |  densify(self)
 |      Convert coefficient matrix to dense array format.
 |
 |      Converts the ``coef_`` member (back) to a numpy.ndarray. This
is the
 |      default format of ``coef_`` and is required for fitting, so
calling
 |      this method is only required on models that have previously
been
 |      sparsified; otherwise, it is a no-op.
 |
 |      Returns
 |      -------
 |      self
 |          Fitted estimator.
 |
 |  sparsify(self)
 |      Convert coefficient matrix to sparse format.
 |
 |      Converts the ``coef_`` member to a scipy.sparse matrix, which
for
 |      L1-regularized models can be much more memory- and storage-
efficient
 |      than the usual numpy.ndarray representation.
 |
 |      The ``intercept_`` member is not converted.
 |
 |      Returns
 |      -------
 |      self
 |          Fitted estimator.
 |
 |      Notes
 |      -----
 |      For non-sparse models, i.e. when there are not many zeros in
``coef_``,
 |      this may actually *increase* memory usage, so use this method
with
 |      care. A rule of thumb is that the number of zero elements,
which can
```

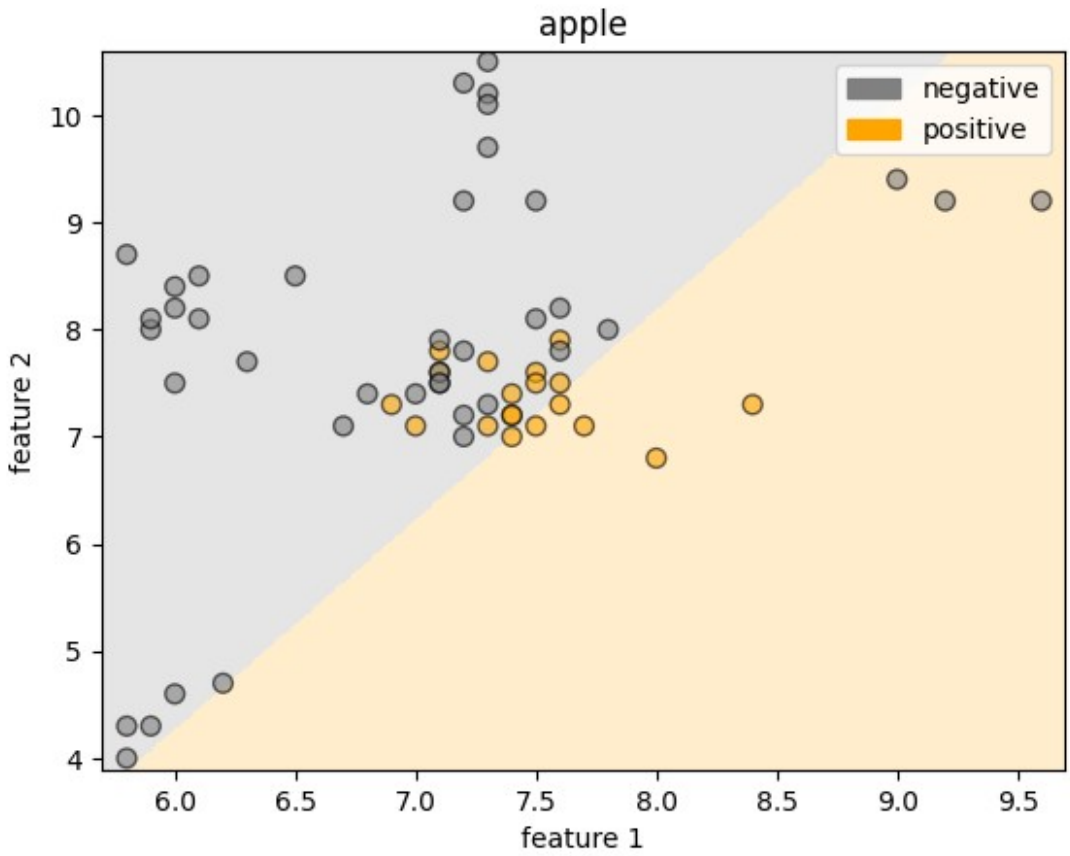
```
 |      be computed with ``(coef_ == 0).sum()``, must be more than 50%
for this
 |      to provide significant benefits.
 |
 |      After calling this method, further fitting with the
partial_fit
 |      method (if any) will not work until you call densify.
 |
 |
----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |      Helper for pickle.
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  __sklearn_clone__(self)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : dict
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested
objects
 |      (such as :class:`~sklearn.pipeline.Pipeline`). The latter have
 |      parameters of the form ``<component>__<parameter>`` so that
it's
 |      possible to update each component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
```

```
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : estimator instance
 |          Estimator instance.
 |
 |
----------------------------------------------------------------------
 |  Methods inherited from
sklearn.utils._metadata_requests._MetadataRequester:
 |
 |  get_metadata_routing(self)
 |      Get metadata routing of this object.
 |
 |      Please check :ref:`User Guide <metadata_routing>` on how the
routing
 |      mechanism works.
 |
 |      Returns
 |      -------
 |      routing : MetadataRequest
 |          A :class:`~sklearn.utils.metadata_routing.MetadataRequest`
encapsulating
 |          routing information.
 |
 |
----------------------------------------------------------------------
 |  Class methods inherited from
sklearn.utils._metadata_requests._MetadataRequester:
 |
 |  __init_subclass__(**kwargs) from builtins.type
 |      Set the ``set_{method}_request`` methods.
 |
 |      This uses PEP-487 [1]_ to set the ``set_{method}_request``
methods. It
 |      looks for the information available in the set default values
which are
 |      set using ``__metadata_request__*`` class attributes, or
inferred
 |      from method signatures.
 |
 |      The ``__metadata_request__*`` class attributes are used when a
method
 |      does not explicitly accept a metadata through its arguments or
if the
 |      developer would like to specify a request value for those
metadata
 |      which are different from the default ``None``.
```

```
 |
 |      References
 |      ----------
 |      .. [1] https://www.python.org/dev/peps/pep-0487
```

```
fruit_label = 1
y_one_vs_all = y==fruit_label
y_one_vs_all = y_one_vs_all.astype(int)

clf = LogisticRegression(C=1000).fit (X, y_one_vs_all)
print ('Accuracy={}'. format (clf.score(X,y_one_vs_all)))
plt.figure()
plot_decision_boundary(clf, X, y_one_vs_all,
title=fruits_dict[fruit_label], precision = 0.01)
```

```
Accuracy=0.7288135593220338
Call prediction for all grid values (precision of drawing = 0.01,
 you may configure to speed up e.g. precision=0.05)
```



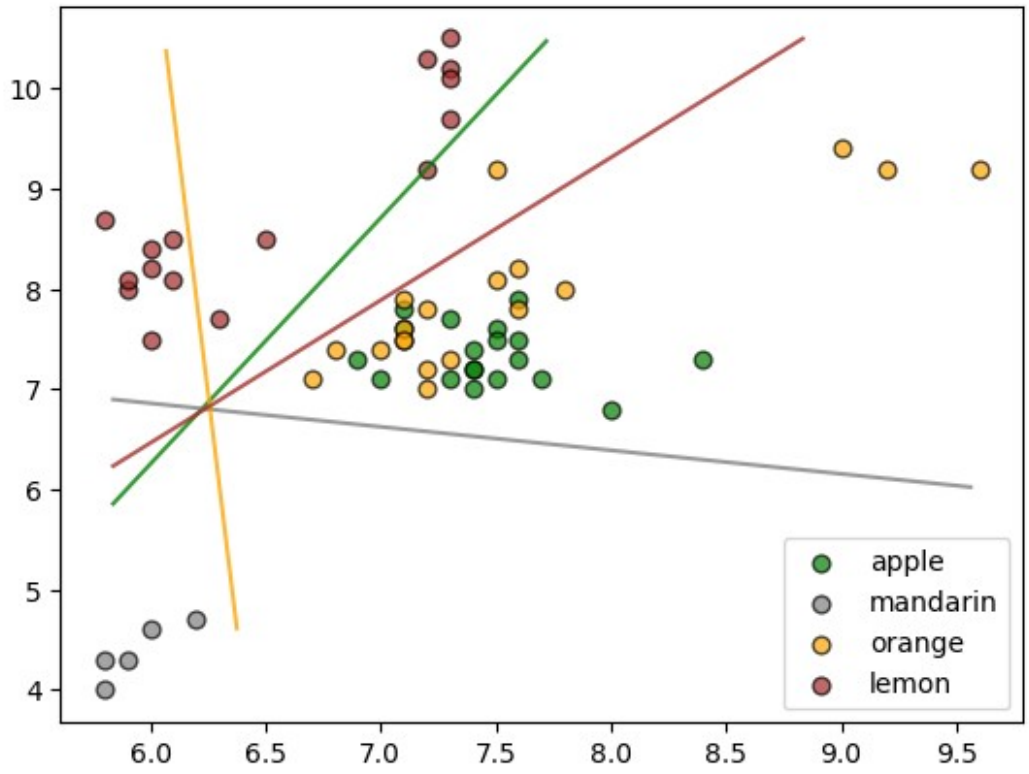
Sklearn multiclass classificcation

```
print ('set(y) = {}'.format (set(y)))
print ('X.shape= {}\ny.shape = {}, '.format (X.shape, y.shape))
clf= LogisticRegression(C=1000, max_iter=2000).fit(X, y)
print ('Accuracy={}'. format (clf.score(X,y)))
clf.intercept_, clf.coef_
```

```
set(y) = {1, 2, 3, 4}
X.shape= (59, 2)
y.shape = (59,),
Accuracy=0.847457627118644
```

```
(array([-29.75018322,  51.86931743, -41.76757778,  19.64844358]),
 array([[  8.61970755,  -3.51258454],
        [ -1.47288929,  -6.27247773],
        [  6.30769266,   0.33620857],
        [-13.45451092,   9.4488537 ]]))
```

```
plt.figure()
plot_multi_class_logistic_regression (X,y,dict_names=fruits_dict)
draw_linear_decision_boundaries_multiclass(clf,X)
```



Iris dataset

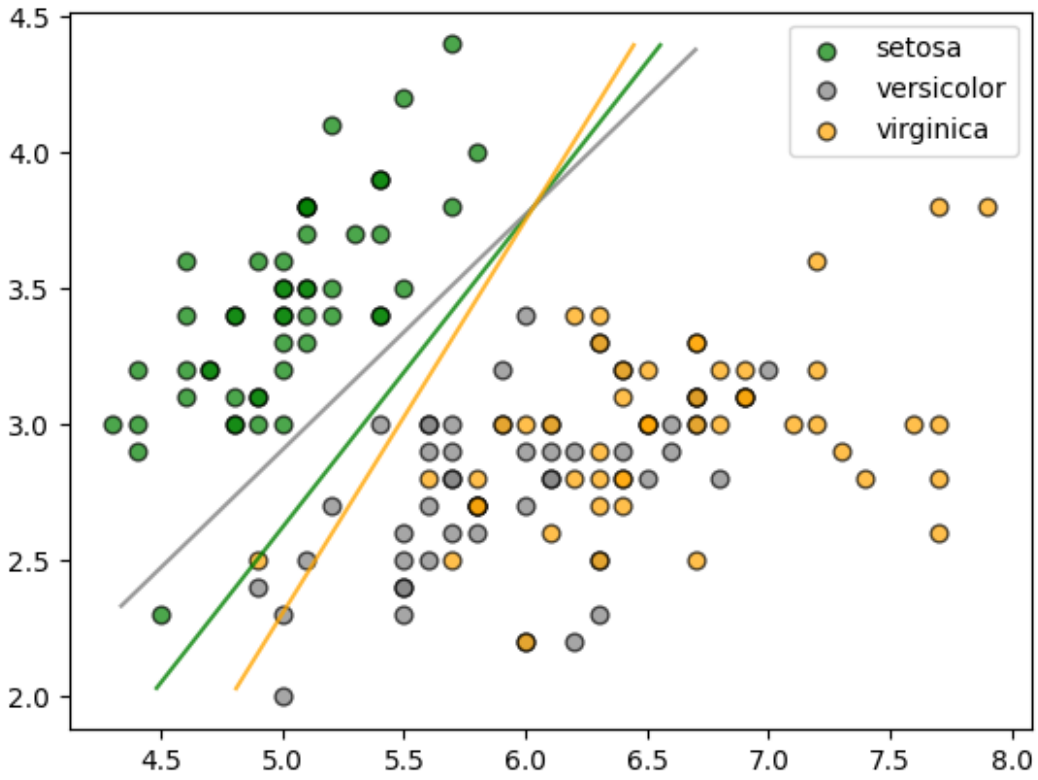
```
from sklearn.datasets import load_iris
iris = load_iris()
X, y, labels = iris.data, iris.target, iris.target_names
print (labels)
clf= LogisticRegression(C=100, max_iter=2000).fit(X, y)
print ('Accuracy={}'. format (clf.score(X,y)))
clf.intercept_, clf.coef_
```

```
['setosa' 'versicolor' 'virginica']
Accuracy=0.98
```

```
(array([ 19.94628563,   5.23090235, -25.17718798]),
 array([[-0.39380232,  3.41248293, -6.40562604, -3.50688137],
        [ 1.35772093,  0.44540853, -0.51275436, -4.41520581],
        [-0.96391861, -3.85789146,  6.9183804 ,  7.92208718]]))
```

```
features = ['sepal_length', 'sepal_width', 'petal_length',
'petal_width']
dict_names = {i:v for i,v in enumerate(labels)}

X2= X[:,:2]
clf= LogisticRegression(C=100).fit(X2, y)
print ('Accuracy={}'. format (clf.score(X2,y)))
print ('clf.intercept_={}, \nclf.coef_=\n{}'.format(clf.intercept_,
clf.coef_))
plt.figure()
plot_multi_class_logistic_regression (X2,y, dict_names = dict_names)
draw_linear_decision_boundaries_multiclass(clf,X2)
```

```
Accuracy=0.8333333333333334
clf.intercept_=[ 25.20698869  -6.10712004 -19.09986865],
clf.coef_=
[[-9.33565746  8.18421204]
 [ 3.72540037 -4.30414546]
 [ 5.61025709 -3.88006658]]
```


setosa
versicolor
virginica
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0