Linear_classification_wine_dataset

December 21, 2023

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

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE" and remove every line containing the expression: "raise …" (if you leave such a line your code will not run).

Do not remove any cell from the notebook you downloaded. You can add any number of cells (and remove them if not more necessary).

0.1 IMPORTANT: make sure to rerun all the code from the beginning to obtain the results for the final version of your notebook, since this is the way we will do it before evaluating your notebook!!!

Fill in your name and id number (numero matricola) below:

[1]:
```
NAME = "Tommaso Bergamasco"
ID_number = int("2052409")

import IPython
assert IPython.version_info[0] >= 3, "Your version of IPython is too old,␣
 ↪please update it."
```

---

1 Classification on Wine Dataset

1.0.1 Dataset description

We will be working with a dataset on wines from the UCI machine learning repository (http://archive.ics.uci.edu/ml/datasets/Wine). It contains data for 178 instances. The dataset is the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines.

1.0.2 The features in the dataset are:

- Alcohol
- Malic acid
- Ash

- Alcalinity of ash
- Magnesium
- Total phenols
- Flavanoids
- Nonflavanoid phenols
- Proanthocyanins
- Color intensity
- Hue
- OD280/OD315 of diluted wines -Proline

We first import all the packages that are needed

[2]:
```
%matplotlib inline
import matplotlib.pyplot as plt


import numpy as np
import scipy as sp
from scipy import stats
from sklearn import datasets
from sklearn import linear_model
import copy
```

[3]:
```
np.random.seed(ID_number)
```

2 Perceptron

We will implement the perceptron and use it to learn a halfspace with 0-1 loss.

Load the dataset from scikit learn and then split in training set and test set (50%-50%) after applying a random permutation to the dataset.

[4]:
```
# Load the dataset from scikit learn
wine = datasets.load_wine()
# Get input and output data from the dataset
X = wine.data
Y = wine.target
# Create new labels
Y = np.where(Y == 0, -1, Y)
Y = np.where(Y == 2, -1, Y)
# Let's get the number of features
d = X.shape[1]
```

[5]:
```
##### Helper functions, do not modify them. You will need them for the first␣
 ↪TODO
def check_constraints(labels, all_possibile_labels, min_num_istances):
    # Count the number of occurrences using numpy
    unique, counts = np.unique(labels, return_counts=True)
    if len(all_possibile_labels) != len(unique):
        return True
    if (counts >= min_num_istances).all():
        return False
    else:
        return True

def need_new_shuffle(y_train, y_test, all_possibile_labels, min_num_istances):
    return (check_constraints(y_train, all_possibile_labels, min_num_istances)␣
 ↪or
            check_constraints(y_test, all_possibile_labels, min_num_istances))
```

[6]:
```
# TODO 1
# Write a function (create_train_val_test_datasets) which takes as input a␣
 ↪dataset and returns 2 datasets:
# S_t and S_test (different runs are supposed) to return different datasets.
# Write a function (create_train_val_test_datasets_with_constraints) which␣
 ↪splits our data in S_t and S_test with
# the additional constraint that in each dataset we MUST have more than␣
 ↪min_num_istances per class.
# Each dataset is represented as a matrix m \times d (numpy ndarray), where m␣
 ↪is the number of data and d is the
# number of features.
# To solve this TODO use the functions we provide you: check_constraints and␣
 ↪need_new_shuffle

def create_train_val_test_datasets(features : np.ndarray, labels: np.ndarray,␣
 ↪m_t : int, m_test : int):
    '''
    Create training (S_t) and test (S_test) sets starting from a dataset.
    This function shuffles the complete dataset before creating the subsets.
    If you call this function twice it is expected to get different S_t, S_test.
 ↪

    :param features: NumPy ndarray containing all the input data data we can use
    :param labels: NumPy ndarray containing all the labels we have
    :param m_t: Number of samples for the training dataset
    :param m_test: Number of samples for the test dataset

    :returns: (x_train, y_train, x_test, y_test)
    :rtype: tuple
        WHERE
        x_train : np.ndarray features in the training dataset
        y_train : np.ndarray labels in the training dataset
        x_test : np.ndarray features in the test dataset
        y_test : np.ndarray labels in the test dataset
    '''
    # SUGGESTION: Use the function np.random.permutation (see the␣
 ↪documentation) to create a permutation of the
    #               dataset indexes. Then use these shuffled indexes to create␣
 ↪S_t, S_val, S_test
    # YOUR CODE HERE

    # To deal with the fact that Y has shape (178,)
    labels = labels.reshape((-1,1))

    # Creation of the matrix [X|Y] of dim(178,14)
    data = np.hstack((features,labels))

    # Permutation
    data = np.random.permutation(data)

    # Creation of the arrays
    x_train = data[0:m_t,0:-1]
    y_train = data[0:m_t,-1]
    x_test = data[m_t:,0:-1]
    y_test = data[m_t:,-1]

    return x_train, y_train, x_test, y_test


def create_train_val_test_datasets_with_constraints(features : np.ndarray,␣
 ↪labels: np.ndarray, m_t : int,
                                                                m_test : int,␣
 ↪min_num_istances : int):
    '''
    Same as function above but now we are imposing the constraints: the␣
 ↪splitted datasets are assumed to contain
    at least min_num_istances per class.

    ...
    :param min_num_istances: Minimum number of istances per class in each of␣
 ↪the splitted datasets
    ...

    '''
    all_possibile_labels = np.unique(labels)
    # YOUR CODE HERE

    # Boolean var. to know when we are done
    shuffle_again = True
    # Repeat the shuffle till we meet the constraints
    while shuffle_again == True:
        x_train, y_train, x_test, y_test =␣
↪create_train_val_test_datasets(features, labels, m_t, m_test)
        shuffle_again = need_new_shuffle(y_train, y_test, all_possibile_labels,␣
↪min_num_istances)

    return x_train, y_train, x_test, y_test

m_t = 80
x_train, y_train, x_test, y_test =␣
↪create_train_val_test_datasets_with_constraints(X, Y, m_t, len(Y)-m_t, 25)
```

[7]:
```
assert x_train.shape == (m_t,       x_train.shape[1]) # here we are comparing␣
↪two tuples (it is an element wise comparison)
assert x_test.shape  == (len(Y)-m_t, x_test.shape[1])
```

[8]:
```
# Let's add a 1 in front of each sample so that we can use a vector to describe␣
↪all the coefficients of the model.
# Do not run this cell multiple times otherwise you will continue adding ones...
↪
# (we add the assert to avoid such issue)

assert x_train.shape[1] == d
assert x_test.shape[1] == d

x_train = np.hstack((np.ones((x_train.shape[0],1)), x_train))
x_test  = np.hstack((np.ones((x_test.shape[0],1)),  x_test))
```

TO DO 2 Now complete the function *perceptron*. Since the perceptron does not terminate if the data is not linearly separable, your implementation should return the desired output (see below) if it reached the termination condition seen in class or if a maximum number of iterations have already been run, where 1 iteration corresponds to 1 update of the perceptron weights. If the perceptron returns because the maximum number of iterations has been reached, you should return an appropriate model (the best seen along the iterations).

The input parameters to pass are: - X: the matrix of input features, one row for each sample - Y: the vector of labels for the input features matrix X - *max_num_iterations*: the maximum number of iterations for running the perceptron

The output values are: - *best_w*: the vector with the coefficients of the best model - *best_error*: the *fraction* of missclassified samples for the best model - *w_iters*: a list of the coefficients found by the algorithm, in those iterations in which the error decreases. This is an 'auxiliary output' (it is not needed for the actual algorithm) that will allows us to have a better insight on the algorithm's behaviour - *error_iters*: a list of the *fractions* of missclassified samples for every model found through the same iterations as for the list above. Again an auxiliary output.

Auxiliary functions

In order to correclty complete the perceptron function it is warmly recommended to define some auxiliary functions ("*find_missclassified*" and "*choose_missclassified*").

"**find_missclassified**" This function looks for missclassified data points in the dataset X.

The input parameters to pass are: - X: the matrix of input features, one row for each sample - Y: the vector of labels for the input features matrix X - *curr_w*: the current value of the parameter vector w

The output value is: - $missclassified\_indeces$: a numpy array cointaining all the missclassified indeces

"**choose_missclassified**" This function return one single index choosen from a array of indeces. If the array is empty it returns a non valid index: -1.

The input parameters to pass are: - $missclassified\_indeces$: numpy arrya containing missclassified indeces

The output value is: - $index$: Integer (or np.int64) containing the choosen index

[9]:
```
# TODO 2

def find_missclassified(X,Y,curr_w):
    # Here you can use np.argwhere to find which model predictions are correct␣
 ↪(this is faster than a for loop)
    # but be carefull on the dimensions of your predictions vector and Y vector.
    # YOUR CODE HERE

    # Reshape Y which is (178,) and curr_w in order to correctly multiply them
    Y = Y.reshape((-1,1))
    curr_w = curr_w.reshape((-1,1))

    # Find missclassified indeces (s.t. y_i<curr_w,x_i> <= 0)
    missclassified_indeces = np.argwhere((X @ curr_w)*Y <= 0)

    # Every missclassified point will be represented by two indeces (row,0)
    # We are interested only in the row's index
    missclassified_indeces = missclassified_indeces[:,0]

    return missclassified_indeces.reshape(-1,)

def choose_missclassified(missclassified_indeces):
    # YOUR CODE HERE
    index = np.random.choice(missclassified_indeces)
    return index

def perceptron(X,Y,max_num_iterations):
    #INITIALIZATION
    curr_w = np.zeros(X.shape[1])
    best_w = curr_w
    num_samples = X.shape[0]
    best_error = num_samples+1  # max + 1 number of possible errors
    w_iters = []
    error_iters = []

    missclassified_indeces = None   # You need to assign this variable the␣
↪proper value
    num_missclassified = None       # You need to assign this variable the␣
↪proper value
    index_missclassified = None     # You need to assign this variable the␣
↪proper value

    # YOUR CODE HERE
    missclassified_indeces = find_missclassified(X,Y,curr_w)
    num_missclassified = len(missclassified_indeces)
    index_missclassified = choose_missclassified(missclassified_indeces)

    num_iter = 1
    while (num_iter <= max_num_iterations and len(missclassified_indeces) != 0):
↪ # Remove this False and pass and place the right termination conditionS
        # Update rule
        # YOUR CODE HERE
        num_iter += 1
        curr_w = curr_w + Y[index_missclassified] * X[index_missclassified,:]

        # Update missclassified data points and choose a new missclassified␣
↪data point
        # YOUR CODE HERE
        missclassified_indeces = find_missclassified(X,Y,curr_w)

        # Check if there exist missclassified indeces
        if len(missclassified_indeces) == 0:
            break

        # And now we are sure we can actually find a missclassified index:
        index_missclassified = choose_missclassified(missclassified_indeces)
        num_missclassified = len(missclassified_indeces)

        # Update (if necessary) the best error achieved together with the best␣
↪parameter up to now.
        # Use copy.copy(curr_w) to copy your current w on "best_w" (since these␣
↪are arrays you would be copying
        # only the pointer if you do not use copy.copy) and to append your␣
↪current w to the list of all w found so far
        # Append the current error to the list of all errors found so far
        # YOUR CODE HERE
        # Check if we found a better solution and update values
        if num_missclassified < best_error:
            # Update the current w and current error
            best_w = copy.copy(curr_w)
            best_error = num_missclassified

            # Append the values of interest in the 2 lists
            w_iters.append(copy.copy(best_w))
            error_iters.append((best_error))

    best_error = float(best_error)/float(num_samples)
    return best_w, best_error, w_iters, error_iters
```

[10]:
```
index = choose_missclassified(np.array(list(range(100))))
assert type(index) == np.int64 or type(index) == int or type(index) == np.int32
```

[11]:
```
# Now run the perceptron for 100 iterations
# We want just to see the output of the algorithm, so we can avoid assigning␣
 ↪the auxiliary outputs to actual variables
w_found, training_error, _, _ = perceptron(x_train, y_train, 100)
print("Training error with 100 iterations: " + str(training_error))
```

```
Training error with 100 iterations: 0.2625
```

[12]:
```
# TODO 3
# Write a function to compute the fraction of missclassified samples given two␣
 ↪nd.array vectors of shape number of
# data times 1 (column vectors)
def classification_loss(y_target : np.ndarray, predictions : np.ndarray) ->␣
 ↪float:
    '''
    This function computes the fraction of missclassified samples given two␣
 ↪vectors: true labels and predictions.
    :param y_target: output labels
    :param predictions: predictions

    :return: Fraction of missclassified samples
    '''
    # YOUR CODE HERE

    # We have correct calssification when dist(y_target,predictions)==0
    distance = y_target - predictions
    missclassified_indeces = np.argwhere(distance != 0)

    return missclassified_indeces.shape[0] / y_target.shape[0]
```

```
# Write a function to compute the fraction of missclassified samples for a␣
↪generic dataset given inputs, targets and
# a vector w.
def compute_fraction_missclassified(X : np.ndarray, Y : np.ndarray, w : np.
↪ndarray) -> float:
    '''
    This function computes the fraction of missclassified samples of model␣
↪parametrized by w on the data X w.r.t.
    targets Y.
    :param X: input locations
    :param Y: targets
    :param w: parameters of the model to be tested

    :return: Fraction of missclassified samples
    '''
    # YOUR CODE HERE

    # Find the predictions of the model
    predictions = X @ w

    # Note that such predictions are just real numbers which we must rename␣
↪with the
    # right label in Y (1 or -1 for the classification problem seen above)
    predictions = np.where(predictions < 0, -1, 1)

    # Now we are in the exact same case of the previous function:
    fraction_missclass = classification_loss(Y, predictions)

    return fraction_missclass
```

[13]:
```
assert classification_loss(np.array([[1],[3]]), np.array([[1],[3]])) == 0
assert classification_loss(np.array([[1],[2]]), np.array([[1],[3]])) == 0.5
y_labels, y_predictions = np.random.choice(10, 1000000), np.random.choice(10,␣
↪1000000)
assert np.isclose(classification_loss(y_labels, y_predictions), 0.9, atol=0.01)
assert training_error == compute_fraction_missclassified(x_train, y_train,␣
↪w_found)
```

We now use the best model *w_found* to predict the labels for the test dataset and print the fraction of missclassified samples in the test set (that is an estimate of the true loss).

[14]:
```
print(f"Training Error for 100 iterations is␣
↪{compute_fraction_missclassified(x_train, y_train, w_found):.4f}")
print(f"Test Error for 100 iterations is␣
↪{compute_fraction_missclassified(x_test, y_test, w_found):.4f}")
```

```
Training Error for 100 iterations is 0.2625
```

```
Test Error for 100 iterations is 0.2551
```

[15]:
```
# now run the perceptron for 10000 iterations
# This time we assign also the auxiliary outputs! They will be useful later on!
w_found, training_error, w_list, error_list_train = perceptron(x_train,␣
 ↪y_train, 10000)
print(f"Training Error for 10000 iterations is␣
 ↪{compute_fraction_missclassified(x_train, y_train, w_found):.4f}")
print(f"Test Error for 10000 iterations is␣
 ↪{compute_fraction_missclassified(x_test, y_test, w_found):.4f}")
```

```
Training Error for 10000 iterations is 0.1875
Test Error for 10000 iterations is 0.2143
```

TO DO 4: Answer in the next cell (you do not need more than 5-7 lines):

1- Consider 100 iterations: what relation do you observe between the training error and the (estimated) true loss? Is this what you expected? Explain what you observe and why it does or does not conform to your expectations.

2- Consider 10000 iterations, what has changed? Explain what you observe and why it does or does not conform to your expectations.

YOUR ANSWER HERE

Answer 1: We would expect that $L_S(w) < \hat{L}_D(w)$ since we are training the Perceptron upon the Training Set. But when running multiple times the Perceptron this doesn't happen so frequently. This happens for 2 reasons: + When dealing with "small" data sets. + When we perform an "insufficient number" of iterations.

Answer 2: In this case $L_S(w) < \hat{L}_D(w)$, and in general we have now to note that, even when

$$[L_S(w) - \hat{L}_D(w)]_{100\ iterations} < 0$$

then the Generalization Gap is s.t.:

$$[\hat{L}_D(w) - L_S(w)]_{1000\ iterations} > [\hat{L}_D(w) - L_S(w)]_{100\ iterations}$$

This is due to the fact that when increasing the iterations of the Perceptron, the model will fit better and better the training set, leading (possibly) to overfitting.

TO DO 5: We want to have a better understanding of the hidden behaviour of perceptron algorithm. We indeed defined two auxiliary outputs that should allow us to inspect the training process in more depth. In particular, we want to check if the sequence of models' errors found through the different iterations is characterized by a decreasing trend, with respect to both training and test datasets.

The function `perceptron` already returns the sequence of errors, which is ready to be plotted. However, the errors for the test set aren not immediately available: we nonetheless have all different models, thus we can easily compute them!

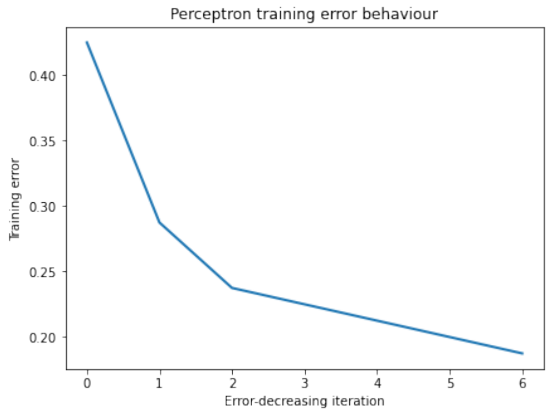
[16]:
```
# TODO 5
# Write a function 'error_models' which takes as input a list of models (each␣
 ↪of which is a numpy ndarray) and returns
# a list of corresponding errors for a generic dataset
# Suggestion: you can use 'compute_fraction_missclassified' from above to solve␣
 ↪this TODO

def error_models(X : np.ndarray, Y : np.ndarray, w_l : list) -> list:
    '''
    This function computes the fraction of missclassified samples of model␣
 ↪parametrized by w on the data X w.r.t.
    targets Y.
    :param X: input locations
    :param Y: targets
    :param w: list of arrays (models) to be tested

    :return: List of fraction of missclassified samples for each model
    '''
    errors = []
    # YOUR CODE HERE
    errors = [compute_fraction_missclassified(X,Y,w) for w in w_l]
    return errors
```

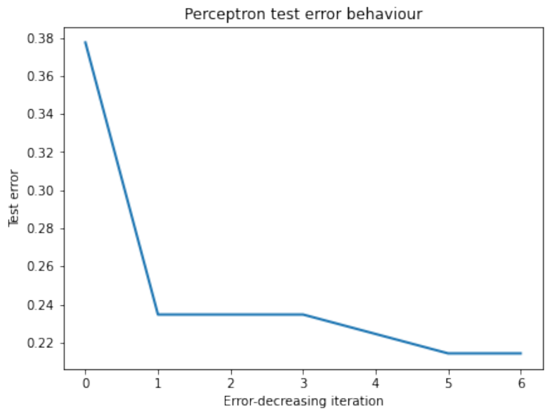
[17]:
```
assert len(error_models(x_test, y_test, w_list)) == len(error_list_train)
assert len(error_models(x_test, y_test, w_list)) <= 10000
```

[18]:
```
# Let's make 'error_list_train' a fraction of the missclassified samples by␣
 ↪dividing for the size of the training set
error_list_train = np.array(error_list_train)/x_train.shape[0]
```

[19]:
```
error_list_test = error_models(x_test, y_test, w_list)
fig, axes = plt.subplots(1, 2, figsize=(15, 5))
axes[0].plot([i for i in range(len(w_list))], error_list_train, linewidth=2)
axes[0].set_xlabel('Error-decreasing iteration')
axes[0].set_ylabel('Training error')
axes[0].set_title('Perceptron training error behaviour')
axes[1].plot([i for i in range(len(w_list))], error_list_test, linewidth=2)
axes[1].set_xlabel('Error-decreasing iteration')
axes[1].set_ylabel('Test error')
axes[1].set_title('Perceptron test error behaviour')
```

[19]:
```
Text(0.5, 1.0, 'Perceptron test error behaviour')
```





TO DO 6: Answer in the next cell (you do not need more than 5-7 lines):

Consider the plot above. How do the two errors compare? Can you identify a particular property of one of them? Why doesn't the other have it?

YOUR ANSWER HERE

The 2 curves are quite similar but we can note that the Training Error, after every iteration, must decrease (Strictly decreasing function by costruction).

On the other hand the Test Error depends on the models w_i obtained to fit the Training Data set and hence we cannot guarantee that the Test Error will decrease at every iteration (even if this would be ideal and it's **almost** what happens when the Training Set is representative of the "true" data).

Note that the Test Error could also increase through iterations.

3 Logistic Regression

Now we use logistic regression, as implemented in Scikit-learn, to predict labels. We first do it for 2 labels and then for 3 labels. We will also plot the decision region of logistic regression.

We first load the dataset again.

[20]:
```
import random
# Let's reinitialize the random seed
random.seed(ID_number)
np.random.seed(ID_number)

# In the following we will keep the dataset with only two classes (which we␣
↪aggregated before)
m_t = 80
x_train, y_train, x_test, y_test =␣
↪create_train_val_test_datasets_with_constraints(X, Y, m_t, len(Y)-m_t, 25)
```

To define a logistic regression model in Scikit-learn use the instruction

$linear\_model.LogisticRegression(C = 1e5, max\_iter = ?)$

C is a parameter related to *regularization*, a technique that we will see later in the course. Setting it to a high value is almost as ignoring regularization, so the instruction above corresponds to the logistic regression you have seen in class. Choose the proper number of iterations: max_iter.

To learn the model you need to use the *fit*(...) instruction and to predict you need to use the *predict*(...) function. See the Scikit-learn documentation for how to use it (have a look at the logreg.score method too).

TO DO Define the logistic regression model, then learn the model using the training set and predict on the test set. Then print the fraction of samples missclassified in the training set and in the test set.

[21]:
```
# TODO 7
# Logistic regression for 2 classes
# To compute the error rate (classification loss you can use the function␣
↪"classification_loss" you built before)
max_iter = None
# YOUR CODE HERE
max_iter = 500 # 100 is the default value, but the algorithm doesn't converge␣
↪and raises
# a warning, setting max_iter to (e.g.) 500 instead converges.

# Define the model and predict using sklearn
log_reg_model = linear_model.LogisticRegression(C = 1e5, max_iter = max_iter).
↪fit(x_train, y_train)
predictions_train = log_reg_model.predict(x_train)
predictions_test = log_reg_model.predict(x_test)

# Errors using classification loss
error_rate_training = classification_loss(y_train, predictions_train)
error_rate_test = classification_loss(y_test, predictions_test)

# Test Error using sklearn implementation (1 - score)
error_rate_test_sklearn = 1 - log_reg_model.score(x_test, y_test)

print("Error rate on training set: "+str(error_rate_training))
print("Error rate on test set: "+str(error_rate_test))
print(f"Compare the estimate of generalization with the sklearn implementation␣
↪{error_rate_test_sklearn}")
```

```
Error rate on training set: 0.0
Error rate on test set: 0.08163265306122448
Compare the estimate of generalization with the sklearn implementation
0.08163265306122447
```

[22]:
```
assert np.isclose(error_rate_test, error_rate_test_sklearn)
```

Now we do logistic regression for classification with 3 classes.

```
[23]: random.seed(ID_number)
np.random.seed(ID_number)

X = wine.data
Y = wine.target

m_t = 80
x_train, y_train, x_test, y_test =␣
 ↪create_train_val_test_datasets_with_constraints(X, Y, m_t, len(Y)-m_t, 20)

_, counts = np.unique(y_train, return_counts=True)
assert (counts >= 20).all()
_, counts = np.unique(y_test, return_counts=True)
assert (counts >= 20).all()
```

```
[24]: # TODO 8
# Logistic regression for 3 classes
# To compute the error rate (classification loss you can use the function␣
 ↪"classification_loss" you built before)
# Choose the proper number of iterations: max_iter.
max_iter = None
# YOUR CODE HERE
max_iter = 300 # this converges with less iterations

# Define the model and predict using sklearn.
# We don't have to change anything: the log_reg's arg multi_class='auto'␣
 ↪recognizes
# which kind of classification we need
log_reg_model = linear_model.LogisticRegression(C = 1e5, max_iter = max_iter).
 ↪fit(x_train, y_train)
predictions_train = log_reg_model.predict(x_train)
predictions_test = log_reg_model.predict(x_test)

# Errors using classification loss
error_rate_training = classification_loss(y_train, predictions_train)
error_rate_test = classification_loss(y_test, predictions_test)

# Test Error using sklearn implementation (1 - score)
error_rate_test_sklearn = 1 - log_reg_model.score(x_test, y_test)

print("Error rate on training set: "+str(error_rate_training))
print("Error rate on test set: "+str(error_rate_test))
print(f"Compare the estimate of generalization with the sklearn implementation␣
 ↪{error_rate_test_sklearn}")
```

```
Error rate on training set: 0.0
Error rate on test set: 0.10204081632653061
Compare the estimate of generalization with the sklearn implementation
0.10204081632653061
```

[25]:
```
assert np.isclose(error_rate_test, error_rate_test_sklearn)
```

TO DO 9: Answer in the next cell (you do not need more than 5-7 lines):

1- Consider logistic regression on 2 and 3 classes what relation do you observe between the training error and the (estimated) true loss in both cases? Is this what you expected? Explain what you observe and why it does or does not conform to your expectations.

2- Consider logistic regression on 2 and perceptron with 10000 iterations, which one would you pick? Do you expect perceptron needs more iterations? Explain what you observe and why it does or does not conform to your expectations.

YOUR ANSWER HERE

Answer 1: Assuming (as it happens in this case) that the data are linearly separable we are always able to find an hypothesis h which makes the Training Error equal to 0 (given a sufficient number of iterations).

Moreover we can note that $\hat{L}_D^{2-class} < \hat{L}_D^{3-class}$ since the 2-class log. reg. problem is obviously "simpler" (only 2 decision regions to determine instead of 3).

Answer 2: The Perceptron **does need** more iterations and, since data are linearly separable, it's able to perfectly solve the separation problem in the Training Set (as the Logistic Regression algorithm).

Nevertheless we can observe that for 10000 iterations L_S and $\hat{L}_D$ are significantly worse than the ones obtained with Log. Reg. and even running the Perceptron with 100000 iterations leads both to a worse performance than the Log. Reg. and to a lot longer running time.

We can hence conclude that the Logistic Regression algorithm is the best pick to solve the problem.

We now are going to plot prediction boundaries of a logistic regression model, in order to plot them we need to reduce the number of features to 2: pick two features and restrict the dataset to include only two features, whose indices are specified in the *feature* vector below. Then split into training and test.

[26]:
```
# TODO 10
#to make the plot we need to reduce the data to 2D, so we choose two features
features_list = ['Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash',␣
 ↪'Magnesium', 'Total phenols', 'Flavanoids',
                 'Nonflavanoid phenols', 'Proanthocyanins', 'Color intensity',␣
 ↪'Hue', 'OD280/OD315 of diluted wines',
                 'Proline']
labels_list = ['class_0', 'class_1', 'class_2']
```

```
index_feature1 = 0  # You can choose the feature you prefer here
index_feature2 = 1  # You can choose the feature you prefer here
features = [index_feature1, index_feature2]

feature_name0 = features_list[features[0]]
feature_name1 = features_list[features[1]]

X = X[:,features]

# In the following we will keep the dataset with 3 classes
m_t = 80
x_train, y_train, x_test, y_test =␣
↪create_train_val_test_datasets_with_constraints(X, Y, m_t, len(Y)-m_t, 20)

# Fit a model on the reduced set of fetures
# YOUR CODE HERE
logreg = linear_model.LogisticRegression(C = 1e5, max_iter=max_iter).
↪fit(x_train, y_train)
```

[27]:
```
assert logreg.predict(x_test).shape == (x_test.shape[0], )
```

The code below uses the model in *logreg* to plot the decision region for the two features chosen above, with colors denoting the predicted value. It also plots the points (with correct labels) in the training set. It makes a similar plot for the test set.

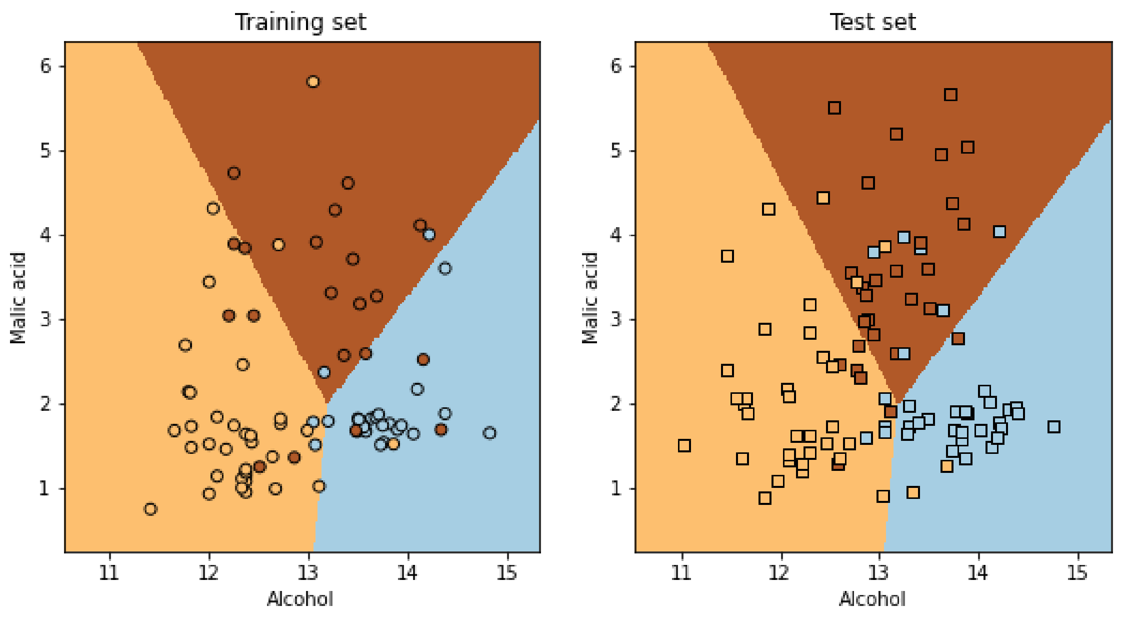
[28]:
```
# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
h = .02  # step size in the mesh
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)

plt.figure(1, figsize=(4, 3))
fig, axes = plt.subplots(1,2, figsize=(10,5))
axes[0].pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
axes[0].scatter(x_train[:, 0], x_train[:, 1], c=y_train, edgecolors='k',␣
↪cmap=plt.cm.Paired)
axes[0].set_xlabel(feature_name0)
axes[0].set_ylabel(feature_name1)
```

```
axes[0].set_xlim(xx.min(), xx.max())
axes[0].set_ylim(yy.min(), yy.max())
axes[0].set_title('Training set')

# Put the result into a color plot
Z = Z.reshape(xx.shape)
axes[1].pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the test points
axes[1].scatter(x_test[:, 0], x_test[:, 1], c=y_test, edgecolors='k', cmap=plt.
 ↪cm.Paired, marker='s')
axes[1].set_xlabel(feature_name0)
axes[1].set_ylabel(feature_name1)

axes[1].set_xlim(xx.min(), xx.max())
axes[1].set_ylim(yy.min(), yy.max())
axes[1].set_title('Test set')
```

```
/tmp/ipykernel_17491/10843995.py:15: MatplotlibDeprecationWarning:
shading='flat' when X and Y have the same dimensions as C is deprecated since
3.3.  Either specify the corners of the quadrilaterals with X and Y, or pass
shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This
will become an error two minor releases later.
  axes[0].pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)
/tmp/ipykernel_17491/10843995.py:28: MatplotlibDeprecationWarning:
shading='flat' when X and Y have the same dimensions as C is deprecated since
3.3.  Either specify the corners of the quadrilaterals with X and Y, or pass
shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This
will become an error two minor releases later.
  axes[1].pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)
```

```
[28]: Text(0.5, 1.0, 'Test set')
```

```
<Figure size 288x216 with 0 Axes>
```



TO DO 11: Answer in the next cell (you do not need more than 5-7 lines):

1- What is the shape of the decision boundaries? Why?

2- In this lower dimensional space, are the features linearily separable? What if you consider the entire feature vector (without any dimensionality reduction)?

YOUR ANSWER HERE

Answer 1: The decision regions are bounded by 3 half-lines.

This happens since the Log. Reg. algorithm solves the 3-class classification problem by solving the three 2-class classification sub-problems and then by returning the intersection of the solutions of such 3 sub-problems.

Answer 2: Considering e.g. the first 2 features, it's easy to see that the algorithm isn't able to separate the data points and since the `logreg = linear_model.LogisticRegression(C = 1e5, max_iter=max_iter)` doesn't raise a `ConvergenceWarning` we can deduce (if it wasn't clear from the plot) that such 2 features aren't linearly separable at all.

When considering the entire feature vector, we are in the 3-class Log. Reg. case of `TODO 8` which achieves $L_S = 0$ meaning that it's possible to separate perfectly the Training data points.

NOTE THAT: We can't say anything about the separability of the Test data points with the info we have. To say something we should run the training phase of the algorithm on the test set and see if we can achieve 0 classification errors.
