DATA LABELING USING LINEAR CLASSIFIERS

1. Objectives

This laboratory is focused on designing a simple linear classifier able to automatically predict the label for unseen data applied as input to the system. The purpose of the laboratory is to help the student gain an intuition of the basic concepts of linear classifiers such as: nearest neighbor classification, k-nearest neighbor classification, data classification with multiclass support vector machine loss. By using the programming language Python the students will implement from scratch a linear classifier that is able to predict the labels for a new set of data (never seen by the system).

2. Theoretical aspects

2.1. The Nearest Neighbor Classifier

The nearest neighbor algorithm was one of the first algorithms used to solve the travelling salesman problem (*i.e.* "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"). In this problem, the salesman starts with a random city and repeatedly visits the nearest city until all have been visited (Fig. 2.1).

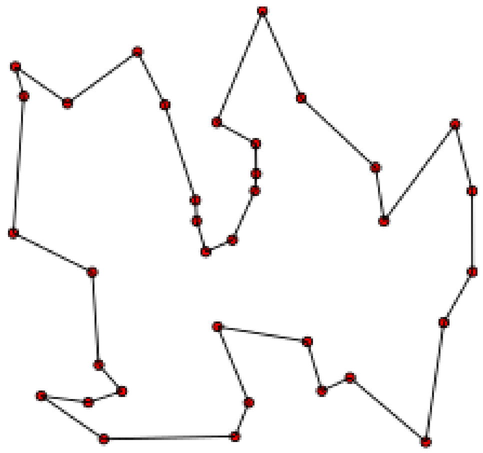


Fig. 2.1. The solution of a travelling salesman problem: the black line shows a possible loop that connects every red dot.

The steps of the nearest neighbor algorithm are presented below:

1. initialize all vertices as unvisited;
2. select an arbitrary vertex, set it as the current vertex ***u***. Mark ***u*** as visited;
3. find out the shortest edge connecting the current vertex ***u*** and an unvisited vertex ***v***;
4. set ***v*** as the current vertex ***u***. Mark ***v*** as visited;
5. if all the vertices in the domain are visited, then terminate. Else, go to step 3.

In this laboratory we will apply the nearest neighbor classifier on one popular image database called ***CIFAR***-10. This dataset consists of 60.000 tiny images that are 32 pixels high and wide. Each image is labeled with one of 10 classes (for example *"airplane, automobile, bird, etc"*). These 60.000 images are partitioned into a training set of 50.000 images and a test set of 10.000 images. Suppose now that we are given the CIFAR-10 training set of 50.000 images (5.000 images for every one of the labels/classes), and we wish to label the remaining 10.000 images. The nearest neighbor classifier will take a test image, compare it to every single one of the training images, and predict the label of the closest training image.

2.2. The k - Nearest Neighbor (k - NN)

It can be noticed that it is strange to only use the label of the nearest image when we wish to make a prediction. Indeed, it is almost always the case that the systems can do better by using what's called a k-Nearest Neighbor Classifier.

The idea is very simple: instead of finding the single closest image in the training set, we will find the ***top k*** closest images, and have them vote on the label of the test image. In particular, when $k = 1$, we recover the Nearest Neighbor classifier. Intuitively, higher values of k have a smoothing effect that makes the classifier more resistant to outliers (Fig. 2.2).

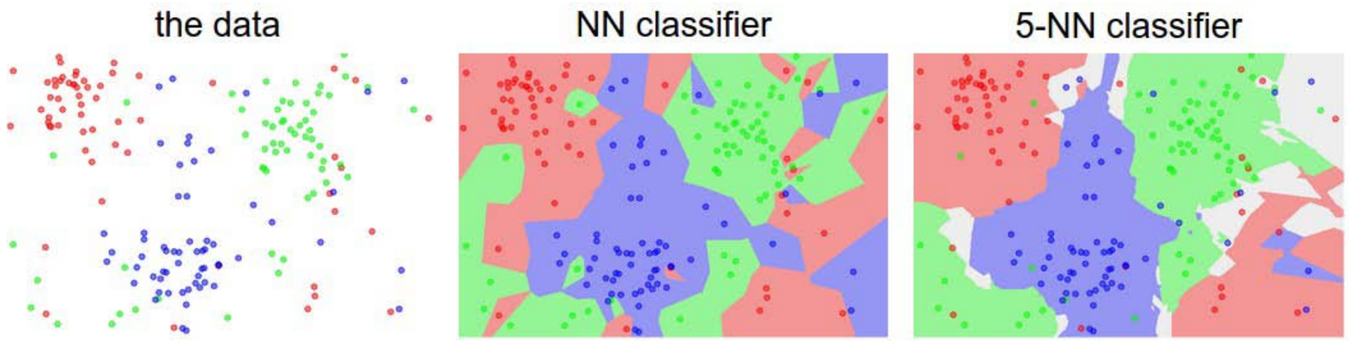


Fig. 2.2. An example of the difference between Nearest Neighbor and a 5-Nearest Neighbor classifier, using 2-dimensional points and 3 classes (red, blue, green).

2.3. Linear Classifiers with SVM loss function

The SVM loss is set up so that the classifier "wants" the correct class for each image to have a score higher than the incorrect classes by some fixed margin Δ. The SVM needs to have a certain outcome in the sense that the outcome would yield a lower loss (which is good).

Let us now get more precise. Recall that for the *i*th example we are given the pixels of image x_i and the label y_i that specify the index of the correct class. The score function takes the pixels and computes the vector $f(x_i, \boldsymbol{W})$ of class scores, which we will abbreviate to $\boldsymbol{s}$ (short for scores). For example, the predicted score for the image x_i as belonging to class j is the j^{th} element in vector $\boldsymbol{s}$: $s_j = f(x_i, W)_j$. The Multiclass SVM loss for the i^{th} example is then formalized as follows:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + \Delta)$$

Our objective will be to find the weights that will simultaneously predict the labels for all examples in the training data and give a total loss that is as low as possible.

Note that in this particular module we are working with linear score functions $f(x_i, W) = \boldsymbol{W} \cdot x_i$, so we can also rewrite the loss function in this equivalent form:

$$L_i = \sum_{j \neq y_i} max(0, w_j^T \cdot x_i - w_{y_i}^T \cdot x_i + \Delta)$$

where w_j is the j^{th} row of $\boldsymbol{W}$ reshaped as a column. However, this will not necessarily be the case once we start to consider more complex forms of the score function $f(\cdot)$. A last piece of terminology that worth mentioning before we finish with this section is that the threshold at zero $max(0,-)$ function is often called the **hinge loss**.

In order to minimize the loss we can compute the gradient with respect to the weights that will tell us the direction in which the function has the steepest rate of increase. However, the gradient does not tell us how far along this direction we should step. Choosing the step size (also called the *learning rate*) will become one of the most important (and most headache-inducing) hyper-parameter setting in training a classifier. In Fig. 2.3 we present the influence of the step size over the loss function.

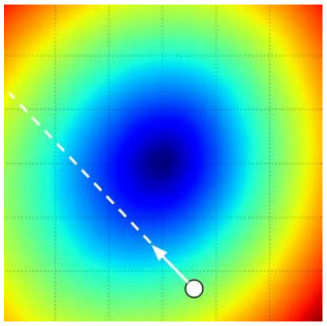


Fig. 2.3. Visualizing the effect of the step size

We start at some particular spot $\boldsymbol{W}$ (a random set of weights) and evaluate the gradient (or rather its negative - the white arrow) which tells us the direction of the steepest decrease in the loss function. Small steps are likely to lead to consistent, but slow progress. Large steps can lead to better progress, but are more risky. Note that eventually, for a large step size we can overshoot the minimum and make the loss worse.

The gradient will be computed analytically based on a direct formula (not numerically approximations). This is very fast to compute, but unlike the numerical gradient it can be more error prone to implement, which is why in practice it is very common to compute the analytic gradient and compare it to the numerical gradient to check the correctness of the implementation. This is called a **gradient check**.

Let us use the example of the SVM loss function for a single data point (x_i):

$$L_i = \sum_{j \neq y_i} max\left(0, w_j^T \cdot x_i - w_{y_i}^T \cdot x_i + \Delta\right)$$

We can differentiate the function with respect to the weights. For example, taking the gradient with respect to w_{y_i} we obtain:

$$\nabla_{w_{y_i}} L_i = -\left(\sum_{j \neq y_i} 1\left(w_j^T \cdot x_i - w_{y_i}^T \cdot x_i + \Delta > 0\right)\right) \cdot x_i$$

where $1(\cdot)$ is the indicator function that is one if the condition inside is true or zero otherwise. While the expression may look scary when it is written out, when implementing this in code the derivative simply count the number of classes that didn't meet the desired margin (and hence contributed to the loss function) and then the data vector x_i scaled by this number is the gradient.

Notice that this is the gradient only with respect to the row of $\boldsymbol{W}$ that corresponds to the correct class. For the other rows where $j \neq y_i$ the gradient is:

$$\nabla_{w_j} L_i = 1\left(w_j^T \cdot x_i - w_{y_i}^T \cdot x_i + \Delta > 0\right) \cdot x_i$$

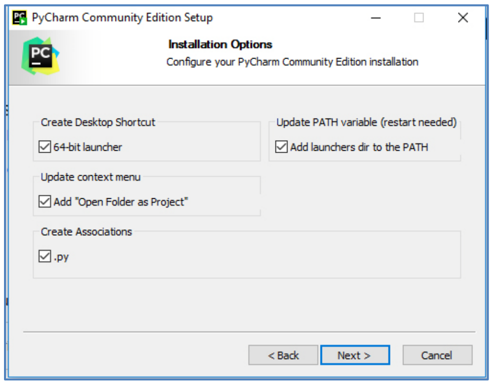
Once you derive the expression for the gradient it is straight-forward to implement the expressions and use them to perform the gradient update.

3. Practical implementation

Pre-requirements: In order to start developing the applications and examples involved in the current laboratory platform in Python with PyCharm you need first to download and install **Python 3.7** from https://www.python.org/downloads/release/python-379/.

For the Windows platforms, during the installation of the Python 3.7 you have to check the box: "Add Python 3.7 to PATH".

The next step is to download the PyCharm IDE from https://www.jetbrains.com/pycharm/download/. Select the **PyCharm Community** version (free and open-source). PyCharm is a dedicated Python Integrated Development Environment (IDE) providing a wide range of essential tools for Python developers, tightly integrated to create a convenient environment for productive Python, web, and data science development. During the PyCharm installation don't forget to check all the required boxes as presented in the Figure 3.1.


PyCharm Community Edition Setup

Installation Options

Configure your PyCharm Community Edition installation

Create Desktop Shortcut
- [✓] 64-bit launcher

Update PATH variable (restart needed)
- [✓] Add launchers dir to the PATH

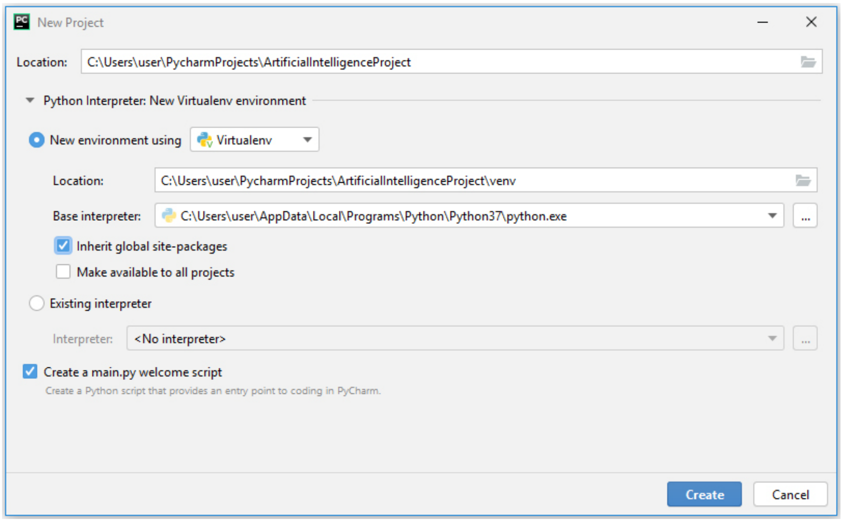
Update context menu
- [✓] Add "Open Folder as Project"

Create Associations
- [✓] .py

Fig. 3.1. PyCharm installation

Observation: Reboot the system as indicated in the last step of the installation.

In **PyCharm** create a new project (File→ NewProject) called ArtificialIntelligenceProject for which a new Environment should be created as in the Figure 3.2. Do not forget to check the box: Inherit global site-packages.


New Project

Location: C:\Users\user\PycharmProjects\ArtificialIntelligenceProject

Python Interpreter: New Virtualenv environment

● New environment using Virtualenv

Location: C:\Users\user\PycharmProjects\ArtificialIntelligenceProject\venv

Base interpreter: C:\Users\user\AppData\Local\Programs\Python\Python37\python.exe

- [✓] Inherit global site-packages
- [ ] Make available to all projects

○ Existing interpreter

Interpreter: <No interpreter>

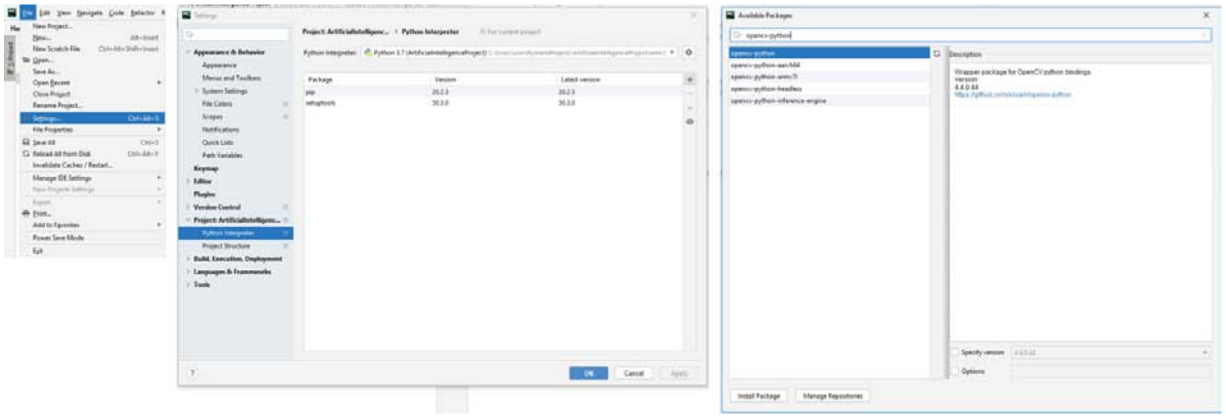
- [✓] Create a main.py welcome script

Create a Python script that provides an entry point to coding in PyCharm.

Fig. 3.2. Defining a virtual environment in PyCharm

In the PyCharm environment install the following dedicated image processing libraries (File → Settings → Project →Python Interpreter → + (Install)) as in Figure 3.3:

- ✓ Opencv-python
- ✓ Numpy
- ✓ Tensorflow
- ✓ Msvc-runtime (only if when importing the above libraries you receive compilation errors)



Project: ArtificialIntelligenc... › Python Interpreter

Python Interpreter: Python 3.7 (ArtificialIntelligenceProject)

Package	Version	Latest version
pip	20.2.3	20.2.3
setuptools	50.3.0	50.3.0

opencv-python

opencv-python
opencv-python-aarch64
opencv-python-armv7l
opencv-python-headless
opencv-python-inference-engine

Wrapper package for OpenCV python bindings
Version
4.4.0.44

Fig. 3.3. Libraries installation in PyCharm

Pre-requirements: Copy the NN_KNN.py script into the project "ArtificialIntelligenceProject" main directory. The current laboratory will use the CIFAR-10 dataset that consists of 60.000 color images with the resolution of 32 x 32 pixels, that are divided in 10 classes (0: airplane, 1: automobile, 2: bird, 3: cat, 4: deer, 5: dog, 6: frog, 7: horse, 8: ship, 9: truck), with 6.000 images per class. There are 50.000 images that can be used for training and 10.000 test images. The following example presents all the steps necessary to develop from scratch a nearest neighbor classifier able to predict the category for any image from the test dataset.

Application 1: Design a Nearest Neighbor classifier able to predict the category for any image from the test dataset. The system will compare the images, pixel by pixel and add up all the differences. In other words, given two images and representing them as vectors I_1 and I_2, a naive choice for comparing them might be the L_1 distance:

$$d_1(I_1, I_2) = \sum_p \left| I_1^p - I_2^p \right|$$

where p represents the total number of pixels of an image. The two images are subtracted element-wise and then all differences are added up to a single number. If two images are identical the result will be zero. But if the images are very different the result will be large.

Step 1: *Load the input data/labels (CIFAR-10) into Python as:*

```
#Input data and Labels

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
```

Each row of the `x_train / x_test` data will contain one image of size 32 x 32 x 3.

Observation: If the following error occurs when loading the data set:

```
<results status="failed" reason="[SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed (_ssl.c:600)" errno="13001"></results>
```

add into the script, the following instruction in the import section:

```
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

Exercise 1: Determine the size of the four vectors (`x_train, y_train, x_test, y_test`) generated after loading the CIFAR-10 dataset.

Exercise 2: Visualize the first 10 images from the testing dataset with their associated labels.

Step 2: *Reshape the training and testing dataset* - The Nearest Neighbor classifier requires the data to be flattened out - one row per image in a 2D array and converted from *uint8* to *float64*. So, in order to compute the differences between two images the training and the testing dataset will be reshaped from a matrix structure to a vector of float elements.

```
np.float64(x_train.reshape(x_train.shape[0], 32 * 32 * 3))
```

Step 3: *Predict the labels for the first 200 images existent in the test dataset* – Each image from the 200 images test dataset will be applied as input to a function in order to obtain the predicted output.

```
for idx, img in enumerate(x_test_flatten[0:200]):
```

In order to perform the image prediction, define a function denoted:
`predictLabelNN(x_train_flatten, y_train, img)` that takes as input three parameters: the reshaped train dataset, the labels of the training dataset and the query image (test image). The function should return as output the predicted label for the current input (`return predictedLabel`).

Within the `predictLabelNN` function the following sub-steps should be performed:

a. Read every image in the training dataset with its associated label;

```
for idx, imgT in enumerate(x_train_flatten):
```

b. Compute the absolute difference between the image from the training dataset and the query image (test image);
c. Add up all differences to a single number (`score`);
d. Determine the nearest image from the training dataset and retain its associated label (`predictedLabel`).

Step 4: *Compare the predicted label for the query image with the ground truth label* – If the prediction is correct than increment the `numberOfCorrectPredictedImages` variable.

Step 5: *Compute the system accuracy* – In order to compute the accuracy use the following relation:

$$Accuracy = 100 * \frac{Nr\_of\_correct\_prediction}{No\_of\_images\_considered\_for\_testing}$$

Exercise 3: Another common choice could be to use the L_2 distance instead of L_1, which has the geometric interpretation of computing the Euclidean distance between two vectors. The distance takes the form:

$$d_2(I_1, I_2) = \sqrt{\sum_p \left(I_1^p - I_2^p\right)^2}$$

In other words we would be computing the pixel-wise difference as before, but this time we square all of them, add them up and finally take the square root. Change the above code in order to determine the image similarity based on the L_2 distance. Determine the accuracy score in this case.

Observation: The scores should be around 35% depending of the images considered for testing. The accuracy of this classifier is better than guessing randomly (10%), but not by much! We could improve things a little by generalizing to a k-nearest neighbors classifier, which instead of finding the closest image to our test set, finds the k-closest images and looks for a consensus in their labels (the dominant class – the class with the highest number of occurrences).

Application 2: Design a K- Nearest Neighbor classifier able to predict the category for any image from the test dataset. The system will compare the images, pixel by pixel and add up all the differences. In other words, given two images and representing them as vectors I_1 and I_2, a naive choice for comparing them might be the L_1 distance:

$$d_1(I_1, I_2) = \sum_p \left|I_1^p - I_2^p\right|$$

where p represents the total number of pixels of an image. The two images are subtracted element-wise and then all differences are added up to a single number. The system will determine the first k-images based on the similarity score and will determine the image label based on the maximum consensus between the classes returned.

Step 1: *Create a new function denoted* `predictLabelKNN(x_train_flatten, y_train, img)` – that takes as input three parameters: the reshaped train dataset, the labels of the training dataset and the query image (test image). The function should return as output the predicted label for the current input (`return predictedLabel`).

Within the `predictLabelKNN` function the following sub-steps should be performed:

a. Read every image in the training dataset with its associated label;

```
for idx, imgT in enumerate(x_train_flatten):
```

b. Compute the absolute difference between the image from the training dataset and the query image (test image);
c. Add up all pixels differences to a single number (`score`);
d. Store all scores and the associated labels (for the test images) in a list denoted: `predictions = []` as pairs (score, label) - `(score, y_train[idx])`;
e. Sort all elements in the `predictions` list in ascending order of priority based on the score parameter;

```
predictions = sorted(predictions, key=lambda x: x[0])
```

f. Retain the top $k = 10$ prediction of the current image;
g. Extract in a separate vector only the labels for the top k predictions;
h. Determine the dominant class (the class with the highest number of occurrences) from the top k prediction list by calling the `most_frequent(list)` function (defined in the NN_KNN.py script).

```
def most_frequent(list):
        list = [x[0] for x in list]
        return [max(set(list), key = list.count)]
```

Step 2: *In the main function replace the* `predictLabelNN` *function with the* `predictLabelKNN` *in order to call the new function.*

Exercise 4: Modify the number of neighbors of the k-NN algorithm as specified in Table 1. What can be observed regarding the system accuracy? How about the prediction time? What will happen if we select $k = 1$?

Table 1. System performance evaluation for various numbers of neighbors

No of neighbors	**3**	**5**	**10**	**20**	**50**
System accuracy					

Exercise 5: Change the above code in order to determine the image similarity based on the L_2 distance. Determine the accuracy score in this case.

Table 2. System performance evaluation for various numbers of neighbors

No of neighbors	**3**	**5**	**10**	**20**	**50**
System accuracy					

Application 3: Design a linear classifier $f(x_i, \boldsymbol{W}) = \boldsymbol{W} \cdot x_i$ that uses the SVM loss function and is able to adjust, the random initialized set of weights $\boldsymbol{W}$, in order to predict the correct label (y_i) for a set of data applied as input (x_i).

Copy the LinearClassifierWithSVMLoss.py script into the project "ArtificialIntelligenceProject" main directory. The application will use as input a dataset with six points in the four dimensional space (labeled as belonging to one of three possible classes) and a set of weights stored within the matrix $(\boldsymbol{W})$. The system will modify the elements of the $\boldsymbol{W}$ matrix so that the prediction $(\boldsymbol{W} \cdot x_i)$ will be consistent with the set of ground truth labels (`y_train`). Finally, after the training (weights optimization), a new set of points (`x_test`) will be applied as input (test points) and the system will predict the class (label) for them.

Step 1: *Consider the following training/testing datasets and the matrix of weights* $\boldsymbol{W}$*:*

```
# Input points in the 4 dimensional space
x_train = np.array([[1, 5, 1, 4],
                    [2, 4, 0, 3],
                    [2, 1, 3, 3],
                    [2, 0, 4, 2],
                    [5, 1, 0, 2],
                    [4, 2, 1, 1]])

# Labels associated with the input points
y_train = [0, 0, 1, 1, 2, 2]

# Input points for prediction
x_test = np.array([[1, 5, 2, 4],
                   [2, 1, 2, 3],
                   [4, 1, 0, 1]])

# Labels associated with the testing points
y_test = [0, 1, 2]

# The matrix of wights
W = np.array([[-1, 2, 1, 3],
              [2, 0, -1, 4],
              [1, 3, 2, 1]])
```

Step 2: *Take each input point and compute the scores* **s** *for all classes.*

```
  for idx, xsample in enumerate(x_train):
```

The scores are computed as $\boldsymbol{s} = \boldsymbol{W} \cdot \boldsymbol{x}_i$:

$$\boldsymbol{s} = \boldsymbol{W} \cdot \boldsymbol{x}_i = \begin{bmatrix} -1 & 2 & 1 & 3 \\ 2 & 0 & -1 & 4 \\ 1 & 3 & 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 5 \\ 1 \\ 4 \end{bmatrix} = \begin{bmatrix} 22 \\ 17 \\ 22 \end{bmatrix}$$

In the above example the point $\boldsymbol{x}_0$ is predicted with score $s_0 = 22$ as belonging to class 0, with the score $s_1 = 17$ to belong to class 1 and with score $s_2 = 22$ to belong to class 2.

In order compute the scores, define a function denoted: `predict(xsample, W),` that takes as input two parameters: a train data point and the weights matrix and returns as output a vector of scores (`return s`).

Step 3: *Compute the loss for each data point (sample)* - The Multiclass SVM loss will be used. For the i^{th} data sample the SVM loss is formalized as follows:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + \Delta)$$

In order compute the loss for a data point define a function denoted: `computeLossForASample(s, labelForSample, delta)` that takes as input three parameters: the score vector `s`, the ground truth label of the current sample `y_train[idx]`, and delta (the SVM margin `delta = 1`) and returns as output the loss (`return loss_i`).

Step 4: *Compute the loss gradient* – The loss function gradient with respect to w_{y_i} can be computed as:

$$\nabla_{w_{y_i}} L_i = -\left(\sum_{j \neq y_i} 1\left(w_j^T \cdot x_i - w_{y_i}^T \cdot x_i + \Delta > 0\right) \right) \cdot x_i$$

where $1(\cdot)$ is the indicator function that is one if the condition inside is true or zero otherwise. Notice that this is the gradient only with respect to the row of $\boldsymbol{W}$ that corresponds to the correct class. For the other rows where $j \neq y_i$ the gradient is:

$$\nabla_{w_j} L_i = 1\left(w_j^T \cdot x_i - w_{y_i}^T \cdot x_i + \Delta > 0\right) \cdot x_i$$

In order to compute the loss gradient define a function denoted:
`computeLossGradientForASample (W, s, currentDataPoint, labelForSample, delta)` that takes as input five parameters: the weights matrix `W`, the scores vector `s`, the current data point `x_train[idx]`, the label for the current data point `y_train[idx]`, and delta (the SVM margin `delta = 1`) and returns as output the loss gradient (`return dW_i`).

```
def computeLossGradientForASample(W, s, currentDataPoint, labelForSample, delta):

    dW_i = np.zeros(W.shape)
    syi = s[labelForSample]

    for j, sj in enumerate(s):
        distance = sj - syi + delta
```

```
            if j == labelForSample:
                continue

            if distance > 0:
                dW_i[j] = currentDataPoint
                dW_i[labelForSample] = dW_i[labelForSample] - currentDataPoint

        return dW_i
```

Step 5: *Compute the global loss (`loss_L`) for all the samples* – The global loss can be computed as the sum of losses `loss_i` determined for all samples.

Step 6: *Compute the global gradient loss matrix* (`dW`) – The global gradient loss matrix can be computed as the sum of gradient losses `dW_i` determined for all samples.

Step 7: *Compute the global normalized loss* – The normalized global loss can be computed as the ratio between the global loss (`loss_L`) and the number of input samples.

Step 8: *Compute the global normalized gradient loss matrix* – The global normalized gradient can be computed as the ratio between the global gradient loss (`dW`) and the number of the input samples.

Step 9: *Adjust the weights matrix* (`W`) – The weights matrix is adjusted using the following relation:

```
W = W - step_size * dW
```

where the `step_size` control the gradient strength over the weights adjustment process.

Exercise 6: Repeat the process, for the new set of weights, until the loss variation is inferior to 0.001. Determine the number of steps necessary for the algorithm to converge?

Exercise 7: Using the set of weights determined at Exercise 6, predict the labels for all the points existent in the `x_test` variable. Compare the results with the ground truth labels (`y_test`). What is the system accuracy?

Note: To retrieve the class with the maximum score `argmax()`function can be used!

Exercise 8: Starting from the code presented in Application 3: "LinearClassifierWithSVMLoss.py" modify the script in order to perform prediction on the Iris Flowers Dataset. The Iris Flowers Dataset involves predicting the flower species given measurements of iris flowers.

It is a multi-category classification problem. The number of observations for each class is balanced. There are 150 observations with 4 input variables and 1 output variable. The variable names are as follows:

1. Sepal length in cm.
2. Sepal width in cm.
3. Petal length in cm.
4. Petal width in cm.
5. Class (Iris Setosa, Iris Versicolour, Iris Virginica).

The data is stored in a *.csv file denoted: “iris.csv”. Read the csv file, ***shuffle*** the data and then split the data into: `x_train, y_train, x_test, y_test.`

From the 150 observations existent in the *.cvs file 120 elements will be used for training and 30 will be used for testing. The weights matrix ***W*** will be initialized with random numbers ranging from 0 to 1.
After performing the system training/evaluation specify:

- What is the optimal value for the weights adjustment step in order to obtain the maximum testing accuracy?
- What is the minimum number of steps necessary to train the system in order to obtain an accuracy superior to 90%?
- Is the system influenced by the random initialization of the weights matrix? Justify your answer?
- Can this system reach a 100% of accuracy in the testing stage?

OBSERVATION: The laboratory report (*.pdf file) will contain:

- The code for all the exercises. The code lines introduced in order to solve an exercise will be marked with a different color and will be explained with comments.
- Print screens with the results displayed in the PyCharm console.
- The exercises solutions: tables of results and responses to the questions.