Q-2 : Multi-class logistic regression

- Akshay Bankar (2019201011)

Softmax regression, also called multinomial logistic regression extends logistic regression to multiple classes.

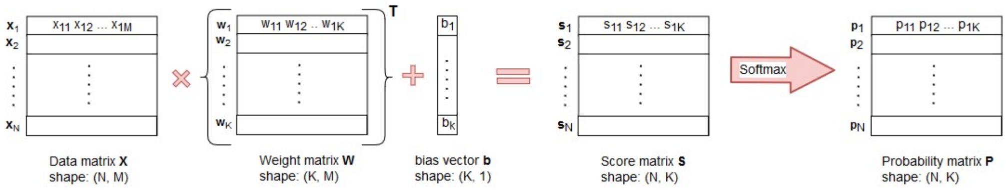
Given:

- dataset $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
- with $x^{(i)}$ being a $d-$dimensional vector $x^{(i)} = (x_1^{(i)}, \ldots, x_d^{(i)})$
- $y^{(i)}$ being the target variable for $x^{(i)}$, for example with $K = 3$ classes we might have $y^{(i)} \in \{0, 1, 2\}$

A softmax regression model has the following features:

- a separate real-valued weight vector $w = (w^{(1)}, \ldots, w^{(d)})$ for each class. The weight vectors are stored as rows in a weight matrix.
- a separate real-valued bias b for each class
- the softmax function as an activation function
- the cross-entropy loss function

An illustration of the whole procedure is given below.



Training steps of softmax regression model :

Step 0: Initialize the weight matrix and bias values with zeros (or small random values).

Step 1: For each class k compute a linear combination of the input features and the weight vector of class k, that is, for each training example compute a score for each class. For class k and input vector $x^{(i)}$ we have:

$$score_k(x^{(i)}) = w_k^T \cdot x^{(i)} + b_k$$

where $\cdot$ is the dot product and $w_{(k)}$ the weight vector of class k. We can compute the scores for all classes and training examples in parallel, using vectorization and broadcasting:

$$scores = X \cdot W^T + b$$

where X is a matrix of shape $(n_{samples}, n_{features})$ that holds all training examples, and W is a matrix of shape $(n_{classes}, n_{features})$ that holds the weight vector for each class.

Step 2: Apply the softmax activation function to transform the scores into probabilities. The probability that an input vector $x^{(i)}$ belongs to class k is given by

$$\hat{p}_k(x^{(i)}) = \frac{\exp(score_k(x^{(i)}))}{\sum_{j=1}^{K} \exp(score_j(x^{(i)}))}$$

Again we can perform this step for all classes and training examples at once using vectorization. The class predicted by the model for $x^{(i)}$ is then simply the class with the highest probability.

Step 3: Compute the cost over the whole training set. We want our model to predict a high probability for the target class and a low probability for the other classes. This can be achieved using the cross entropy loss function:

$$J(W, b) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left[y_k^{(i)} \log(\hat{p}_k^{(i)}) \right]$$

In this formula, the target labels are *one-hot encoded*. So $y_k^{(i)}$ is 1 is the target class for $x^{(i)}$ is k, otherwise $y_k^{(i)}$ is 0.

Step 4: Compute the gradient of the cost function with respect to each weight vector and bias.

The general formula for class k is given by:

$$\nabla_{w_k} J(W, b) = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \left[\hat{p}_k^{(i)} - y_k^{(i)} \right]$$

For the biases, the inputs $x^{(i)}$ will be given 1.

Step 5: Update the weights and biases for each class k:

$$w_k = w_k - \eta \nabla_{w_k} J$$

$$b_k = b_k - \eta \nabla_{b_k} J$$

where η is the learning rate.

Import libraries

In [1]:

```
import numpy as np
import cv2
import glob
from MyPCA import MyPCA
from sklearn.model_selection import train_test_split
```

Define class for multiclass logistic regression with the steps defined above

In [8]:

```
class LogisticRegression:
    def __init__(self, learn_rate = 0.001, num_iters = 100):
        self.learning_rate = learn_rate
        self.n_iters = num_iters
        self.weights = None
        self.bias = None

    def train(self, data, labels):
        self.data = self.add_bias_col(data)
        self.n_samples, self.n_features = self.data.shape
        self.classes = np.unique(labels)
        self.class_labels = {c:i for i,c in enumerate(self.classes)}
        labels = self.one_hot_encode(labels)
        self.weights = np.zeros(shape=(len(self.classes),self.data.shape[1]
        for _ in range(self.n_iters):
            y = np.dot(self.data, self.weights.T).reshape(-1,len(self.class
            ## apply softmax
            y_predicted = self.softmax(y)
            #y_predicted = self.sigmoidfn(y)

            # compute gradients
            dw = np.dot((y_predicted - labels).T, self.data)
            # update parameters
            self.weights -= self.learning_rate * dw
        #print(self.weights)

    def add_bias_col(self,X):
        return np.insert(X, 0, 1, axis=1)

    def one_hot_encode(self, y):
        return np.eye(len(self.classes))[np.vectorize(lambda c: self.class_
    '''
    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_predicted = self._sigmoid(linear_model)
        y_predicted_cls = [1 if i > 0.5 else 0 for i in y_predicted]
        return np.array(y_predicted_cls)
    '''
    def softmax(self, z):
        return np.exp(z) / np.sum(np.exp(z), axis=1).reshape(-1,1)

    def predict(self, X):
        X = self.add_bias_col(X)
        pred_vals = np.dot(X, self.weights.T).reshape(-1,len(self.classes))
        self.probs_ = self.softmax(pred_vals)
        pred_classes = np.vectorize(lambda c: self.classes[c])(np.argmax(se
        return pred_classes
        #return np.mean(pred_classes == y)
```

Read the data images and perform PCA using the class defined in Q-1.

The input images are converted to grayscale and resized to (64,64).

Number of PCA components corresponding to 95% of variance are taken.

In [3]:

```
def read_data(path):
        img_files = glob.glob(path)
        #print(img_files)
        gray_images = []
        labels = []
        for file in img_files:
            img = cv2.imread(file)
            img = cv2.resize(img,(64,64),interpolation=cv2.INTER_AREA) #Nor
            flat_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY).flatten()
            gray_images.append(flat_img)
            lab = ((file.split('/')[-1]).split('_')[0]).lstrip('0')
            if not lab:
                labels.append(0)
            else :
                labels.append(int(lab))
        return np.asarray(gray_images), labels

data, labels = read_data("./dataset/*")
pca = MyPCA(n_components = 0.95)#n_components = 0.95
pca_data = pca.fit(data)
print("Shape of data transformed after performing PCA :",pca_data.shape)
#print(labels)
```

```
Shape of data transformed after performing PCA : (520, 137)
```

In [9]:

```python
from sklearn.metrics import accuracy_score, confusion_matrix, classificatic

train_X, test_X, train_y, test_y = train_test_split(pca_data, labels, trair
print("Shape of train data :",np.shape(train_X))
print("Shape of test data :", np.shape(test_X))
logreg = LogisticRegression()
logreg.train(np.asarray(train_X), np.asarray(train_y))
pred_labels = logreg.predict(np.asarray(test_X))
#print("Accuracy : ",, np.asarray(test_y)))

print ("Confusion-matrix :")
print(confusion_matrix(test_y,pred_labels))
print("Classification-report")
print (classification_report(test_y,pred_labels))
print ("Accuracy score :", accuracy_score(test_y,pred_labels))
```

```
Shape of train data : (416, 137)
Shape of test data : (104, 137)
Confusion-matrix :
[[ 9  0  3  0  0  0  1  0]
 [ 2  8  1  0  0  0  0  0]
 [ 0  0 15  0  0  0  0  0]
 [ 0  1  0  7  1  0  0  1]
 [ 0  0  0  1  9  2  0  0]
 [ 0  0  3  0  1  9  0  0]
 [ 0  0  2  2  0  0  8  1]
 [ 0  0  0  1  1  0  0 15]]
Classification-report
              precision    recall  f1-score   support

           0       0.82      0.69      0.75        13
           1       0.89      0.73      0.80        11
           2       0.62      1.00      0.77        15
           3       0.64      0.70      0.67        10
           4       0.75      0.75      0.75        12
           5       0.82      0.69      0.75        13
           6       0.89      0.62      0.73        13
           7       0.88      0.88      0.88        17

    accuracy                           0.77       104
   macro avg       0.79      0.76      0.76       104
weighted avg       0.79      0.77      0.77       104

Accuracy score : 0.7692307692307693
```