Assignment 2

Mahir Hambiralovic

February 21, 2022

1 Summary

A Computer Exercise on classification of a text dataset was completed. The exercises were to classify two corpuses of text, one in English and one in French of *Salammbô* using linear regression, perceptrons and logistic regression, all using gradient descent.

2 My implementations

2.1 Linear Regression

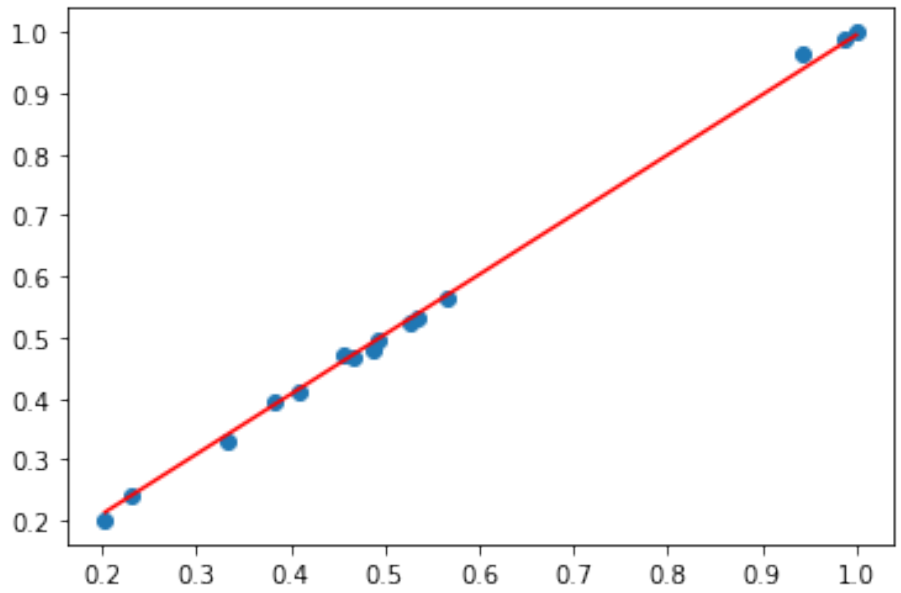
The task was to fit the number of A's in a text, given the total number of characters in the corpus using Linear Regression.

2.1.1 Batch descent

```
def fit_batch(X, y, alpha, w,
                  epochs=500,
                  epsilon=1.0e-5):
    w = np.array(w)
    epoch = 1
    error = 10000

    while error > epsilon and epoch <= epochs:
        (b, m) = w
        y_pred = m*X + b

        loss_m = sum(X * (y - y_pred))
        loss_b = sum(y - y_pred)

        m = m + (alpha / len(y)) * loss_m
        b = b + (alpha / len(y)) * loss_b

        epoch += 1
        error = np.linalg.norm(y - y_pred)
        w = (b, m)

    print('stopped at Error:', error, 'Epoch:', epoch)
    return w

(b, m) = fit_batch(X, y, 0.01, [random.random(), random.random()])
```

2.1.2 Stochastic descent

```
# Write your code here
def fit_stoch(X, y, alpha, w,
                  epochs=500,
                  epsilon=1.0e-5):
    epoch = 1
    error = 10000
    idx = list(range(len(X)))
    while error > epsilon and epoch <= epochs:
        errors = []
        random.shuffle(idx)
        for i in idx:
            x = X[i]
            y_i = y[i]

            (b, m) = w
            y_i_pred = m*x + b

            loss_m = x * (y_i - y_i_pred)
            loss_b = y_i - y_i_pred

            m = m + alpha * loss_m
            b = b + alpha * loss_b

            y_pred = m*X + b
            error = np.linalg.norm(y - y_pred)
            w = (b, m)

        # error = np.linalg.norm(errors)
        epoch += 1
    return w

(b, m) = fit_stoch(X, y, 0.01, [random.random(), random.random()])
```



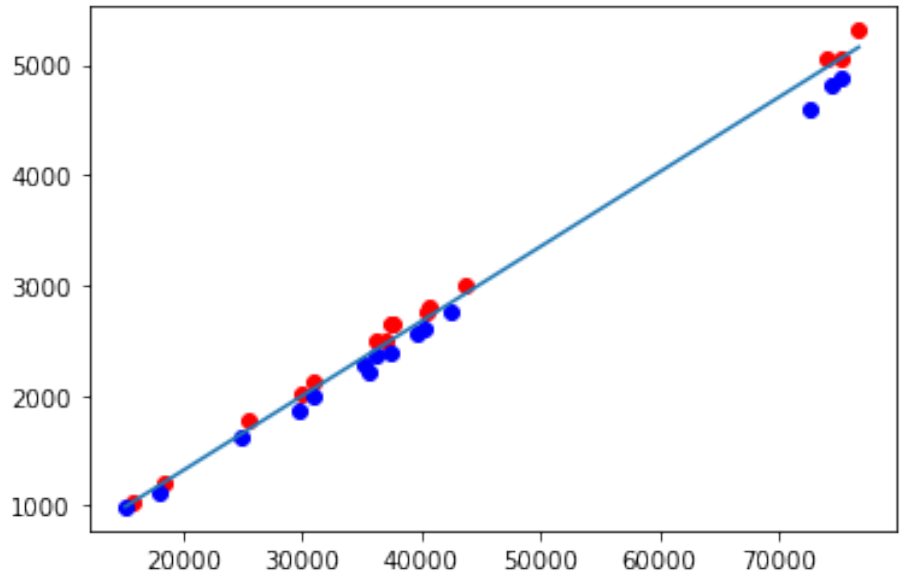
2.2 Perceptron

The task was to classify whether a text was in French or in English based on the number of letter's and the number of A's in the text.

```
def predict(X, w):
    y_pred = np.matmul(w, X)
    if y_pred >= 0:
        return 1
    else:
        return 0

def fit_stoch(X, y,
              alpha=0.01,
              epochs=1000,
              max_misclassified=0,
              verbose=True):
    epoch = 1
    misclassified = 0
    correct_classified = 0

    idx = list(range(len(X)))
    w = [random.random() for _ in range(len(X[0]))]

    while epoch <= epochs:
        if max_misclassified != 0:
            if misclassified > max_misclassified:
                break

        random.shuffle(idx)

        for i in idx:
            x_i = X[i]
            y_i = y[i]

            y_i_pred = predict(x_i, w)
            error = (y_i - y_i_pred)

            for j in range(len(w)):
                w_j = w[j]
                x_j = x_i[j]
                w[j] += alpha*(error)*x_j

            if abs(error):
                misclassified += 1
            else:
                correct_classified += 1

        epoch += 1

    correct_classified, 'epoch', epoch)
    return w

def leave_one_out_cross_val(X, y, fitting_function):
    n = len(X)
    n_correct = 0

    for fold in range(n):
        X_train = X[:fold] + X[fold +1:]
        X_test = X[fold]

        y_train = y[:fold] + y[fold +1:]
        y_test = y[fold]
```

```
59
60          w = fitting_function(X_train, y_train)
61
62          y_pred = predict(X_test, w)
63          correct = not abs(y_test - y_pred)
64          n_correct += correct
65
66          print('Fold', fold+1, 'on', n, ': weights:', w, 'Correct.'
        if correct else 'Wrong.')
67          # print('Fold', fold, 'on 30: epochs:', epochs, 'weights:',
         w)
68      print('Correct', n_correct, '/', n)
69      return n_correct / n
```

With this implementation, a cross validation accuracy of 86.7% was achieved. Restored weights [7.083916039112965, -0.014127309008252075, 0.20878803577563473]. I would consider these good results for practically a linear regression implementation of the ANN (we can see in the decision boundary and the data points in the figure, that a straight line cannot fit the points perfectly). With one or two more perceptrons, we should be able to fit the curve perfectly, but risk overfitting.



2.2.1 Logistic Regression

Implementing the same task as above, but using Logsitic Regression.

```
1  def logistic(x):
2      try:
3          return 1 / ( 1 + math.exp(-x) )
4      except OverflowError:
5          x_rounded = 500 * x/abs(x)
6          return 1 / ( 1 + math.exp(-x_rounded) )
7
8  def predict_proba(X, w):
9      return logistic(np.matmul(w, X))
10
```

```
def predict(X, w):
    y_pred = predict_proba(X, w)
    return 1 if y_pred >= 0.5 else 0

def fit_stoch(X, y, alpha=0.5,
              epochs=4000,
              epsilon=1.0e-4,
              verbose=False):
    epoch = 1

    idx = list(range(len(X)))
    w = [random.random() for _ in range(len(X[0]))]

    while epoch <= epochs:
        random.shuffle(idx)

        for i in idx:
            x_i = X[i]
            y_i = y[i]

            y_i_pred = predict_proba(x_i, w)
            # error = (y_i - y_i_pred)

            for j in range(len(w)):
                # w_j = w[j]
                x_j = x_i[j]
                w[j] += alpha * (y_i - y_i_pred) * y_i_pred * (1 -
    y_i_pred) * x_j


        epoch += 1

    return w

# Write your code here
def leave_one_out_cross_val(X, y, fitting_function):
    n = len(X)
    n_correct = 0

    for fold in range(n):
        X_train = X[:fold] + X[fold +1:]
        X_test = X[fold]

        y_train = y[:fold] + y[fold +1:]
        y_test = y[fold]

        w = fitting_function(X_train, y_train)

        y_pred = predict(X_test, w)
        # print(y_pred)
        correct = not abs(y_test - y_pred)
        n_correct += correct

        print('Fold', fold+1, 'on', n, ': weights:', w, 'Correct.'
    if correct else 'Wrong.')
        # print('Fold', fold, 'on 30: epochs:', epochs, 'weights:',
     w)
    print(n_correct, n)
    return n_correct / n
```

With this implementation, a cross validation accuracy of 86.7% was achieved.
Restored weights [-0.1618444091889996, -0.0005504545172716554, 0.008356119511984951]

3 Improvements

In the above tasks, I decided to use stochastic descent where possible. One improvement to this approach was to use larger batches than 1. Furthermore, as discussed in the perceptron section, the ANN approach could reach a perfect score on the data if it would have been allowed to be more complex. However, this would have let it overfit, especially seeing that we tested on the training data.

4 Dissertation of An overview of gradient descent optimization algorithms

Ruth writes, in his article "An overview of gradient descent optimization algorithms" an overview, summary and motivation of the various approaches to gradient descent, the main algorithm for every deep learning implementation. The main gradient descent variants to chose from are explained in the following sections.

4.1 Gradient descent variants

4.1.1 Batch gradient descent

The traditional and "vanillla" approach to gradient descent, calculating the gradient for the entire dataset and calculating the step to take in each step for a fixed number of epochs. Some draw-backs of this approach is that it is very time consuming, as it must compute the gradient of the entire dataset in each epoch. Also it does not support online-training.

4.1.2 Stochastic gradient descent

A "stochastic" approach for updating the weights by creating a gradient for each training example. This approach does however lead to many overfitted and biased steps, that are each perfect to only one training example.

4.1.3 Mini-batch gradient descent

A compromise of the above two, that predefines a batch-size of data to work with, which is trained upon. Thus online training is supported, faster training, and fewer biased "missteps" are taken.

4.1.4 Other optimization methods

Later in the paper, Ruth discusses a myriad of different methods used to optimize the descent method in various ways. The optimizations are used for e.g. avoiding local minimas, speeding up learning and improve learning on sparse data. The final evaluation of the methods show that there can be a large difference in the speed of convergence between the different methods. While not all optimizations were created equal, all of them seem to have some improvement to the vanilla approach, which Ruth criticizes as being overused in many papers. Ruths' paper concludes that Adam is generally the overall best choice.