Artificial Intelligence - Assignment 2

Szymon Stypa - sz6102st-s

February 17, 2021

1 Overview

The goal of this assignment was to understand, train and evaluate linear classifiers using linear and logistic regression methods with gradient ascent/descent and the perceptron algorithm. This was to be done with a dataset containing the counts of letters and the letter A in the 15 chapters of the French and English versions of Salammbô. To perform these tasks, a jupyter notebook with guidelines and all necessary skeleton code was provided by the course staff. The implementation was done in Python.

2 Solution

The code for the solutions was mainly derived from the theory presented during the machine learning lectures 1-2. As for the linear regression section, it is heavily inspired by, and partly copied from, the existing solution pointed to in the notebook.

2.1 Linear Regression

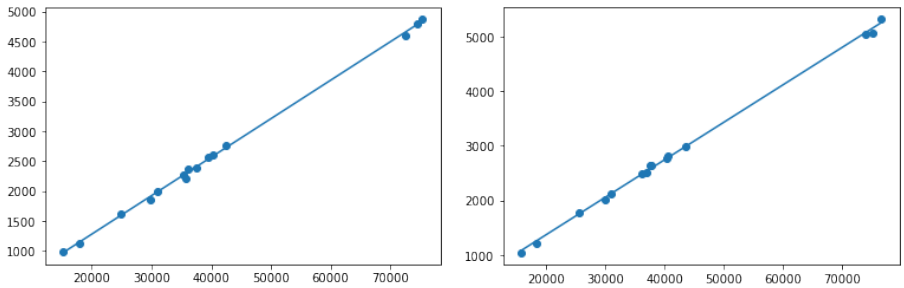
The first task of the assignment consisted of implementing the gradient descent method to compute regression lines fitting the dataset. In other words, the goal is to use gradient descent in order to produce a linear model predicting the count of A's in English and French text. Two variants of this method were to be presented, namely stochastic and batch gradient descent. These differ slightly in implementation and use case, as will be clear after this section. However, ultimately these yield almost the same result when it comes to this assignment. Both versions take the same arguments which besides features and labels consist of the learning rate α, number of epochs and the tolerance ϵ. The functions will loop for the set number of epochs, or until the length of the gradient vector subceeds the tolerance ϵ. Every loop involves adjusting the weights `w` proportionally to α in the direction derived from the gradient of the objective function. This is, as has been mentioned, done slightly differently between the two types of gradient descent.

```
# An epoch in batch descent
loss = y - predict(X, w)
gradient = X.T @ loss
w += alpha * gradient

# An epoch in stochastic descent
for i in idx:
    loss = y[i] - predict(X[i], w)[0]
    gradient = loss * np.array([X[i]]).T
    w += alpha * gradient
```

Here, the `predict` function simply performs a matrix multiplication between the two input values, and `idx` is a shuffled list of all the indices of the dataset. As can be seen, the batch method computes a gradient for the whole dataset and updates weights only once per iteration. The stochastic method, on the other hand, computes the gradient and updates the weights on every data point (provided there is a loss).

Fitting was first done with both types of gradient descent on the English potion of the data set, with 500 epochs, $\alpha = 1$ and $\epsilon = 10^{-5}$. This yielded lower sum of squared error (SSE) values for batch than stochastic descent. Both fits are good, but the lower SSE for batch means a somewhat better fit. This was also the case when running the code on the French potion of the data.



Line regressed using batch descent for the two languages. Left figure is English and right is French.

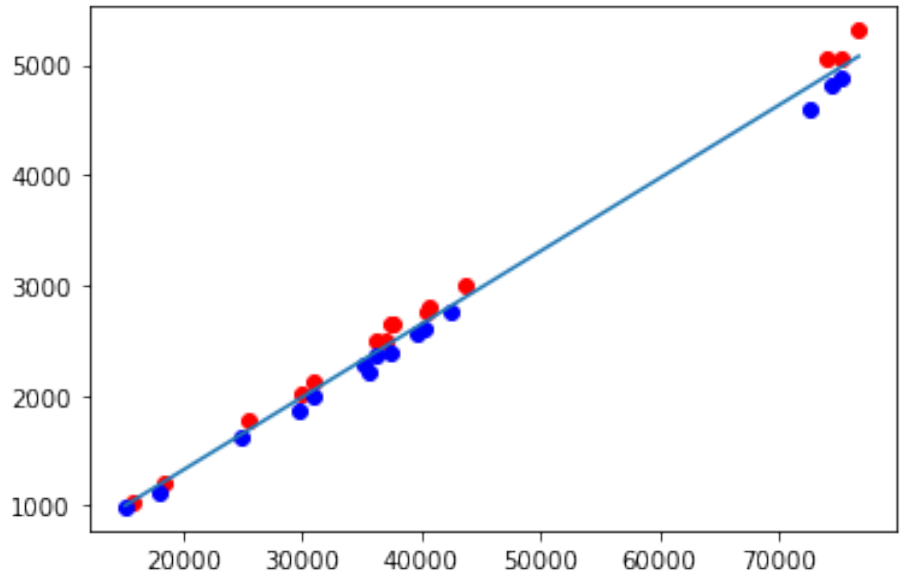
2.2 Perceptron

In this portion of the assignment, the features will consist of the letter count and the A count, whereas the labels will be members of `(0, 1)` representing the two languages. The goal is to train a threshold function for binary linear classification, which can detect the language from the relation between these values. For this purpose, the perceptron algorithm is used along with a new binary prediction function

```
def predict(X, w):
    return 1 if X @ w > 0 else 0
```

The stochastic approach is used for the fitting function, since apparently that is usually the case. Thus, the fitting function resembles the stochastic gradient descent function from before with the exception of the ϵ argument. It

is replaced by a number expressing the maximum misclassified data points per epoch allowed before the fitting is terminated. Using an α value of 1 with 1000 epochs and a misclassification threshold of 0, we get the following result



To evaluate the classification, the leave one out cross validation is introduced. The basic idea of this process is removing one data point from the dataset, fitting on the remaining data and checking wether the obtained weights will yield the right prediction on the removed data. The quality of the algorithm can then be expressed as the number of properly predicted removed data points divided by total number of data points. In the case of this perceptron, that number ended up being approximately 0.97, which is a respectable value meaning that only one removed prediction was misclassified.

2.3 Logistic Regression

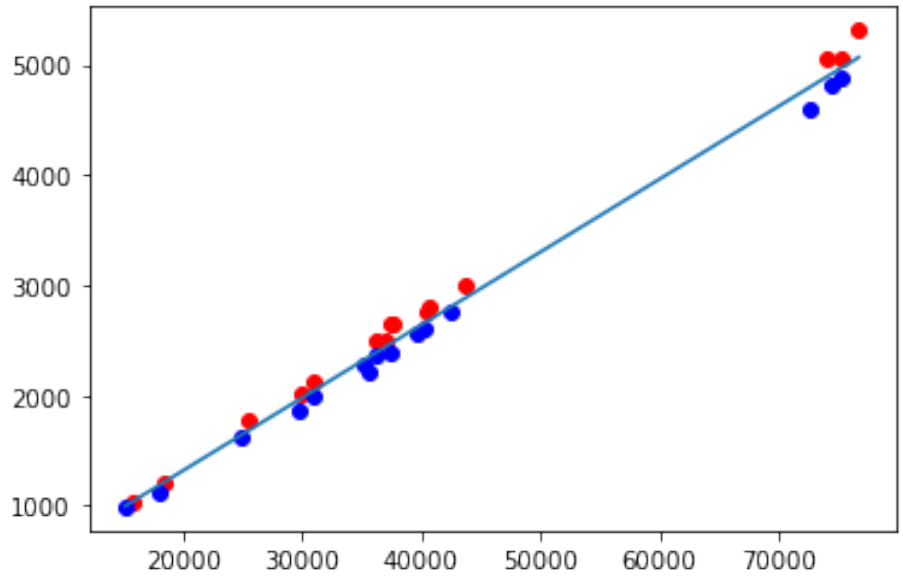
Lastly, logistic regression was used to classify the data. I chose to work with the stochastic version for this part, simply because it seemed more intuitive. The goal here is to use the logistic curve to model the probability of an observation belonging to a class. For this, the simple logistic function is introduced, which will represent the probability of an observation with given weights belonging to class 1 (French).

```
def logistic(x):
    return 1/(1 + math.exp(-x))
```

The fitting function works pretty much like the stochastic descent function for linear regression, except now we're ascending and trying to find the maximum likelihood of classification. This means that the gradient will be derived from the log-likelihood

```
for i in idx:
    loss = y[i] - logistic(X[i] @ w)
    gradient = (X[i] * loss).reshape(-1, 1)
    w += alpha * gradient
```

Besides this, the fitting function remains the same. When fitting is done, the resulting values from the logistic function are interpreted as 1 if the value is larger than 0.5, else 0. With 1000 epochs, α set to 100, and ϵ to 10^{-400}, the ascent seems to converge quite nicely



with a cross validation accuracy of 0.97. Higher ϵ values or lower α values yield very inaccurate models for some reason, but I did not have the time to investigate this thoroughly.

3 Solution Data

3.1 Linear Regression

The data used for the linear regression section consist of two feature-label pairs derived from the data provided in the beginning of the notebook. One for each language:

```
X_en = [[1.0000e+00, 3.5680e+04],
        [1.0000e+00, 4.2514e+04],
        [1.0000e+00, 1.5162e+04],
        [1.0000e+00, 3.5298e+04],
        [1.0000e+00, 2.9800e+04],
        [1.0000e+00, 4.0255e+04],
        [1.0000e+00, 7.4532e+04],
        [1.0000e+00, 3.7464e+04],
```

```
         [1.0000e+00, 3.1030e+04],
         [1.0000e+00, 2.4843e+04],
         [1.0000e+00, 3.6172e+04],
         [1.0000e+00, 3.9552e+04],
         [1.0000e+00, 7.2545e+04],
         [1.0000e+00, 7.5352e+04],
         [1.0000e+00, 1.8031e+04]]

y_en = [[2217],
         [2761],
         [ 990],
         [2274],
         [1865],
         [2606],
         [4805],
         [2396],
         [1993],
         [1627],
         [2375],
         [2560],
         [4597],
         [4871],
         [1119]]

X_fr = [[1.0000e+00, 3.6961e+04],
         [1.0000e+00, 4.3621e+04],
         [1.0000e+00, 1.5694e+04],
         [1.0000e+00, 3.6231e+04],
         [1.0000e+00, 2.9945e+04],
         [1.0000e+00, 4.0588e+04],
         [1.0000e+00, 7.5255e+04],
         [1.0000e+00, 3.7709e+04],
         [1.0000e+00, 3.0899e+04],
         [1.0000e+00, 2.5486e+04],
         [1.0000e+00, 3.7497e+04],
         [1.0000e+00, 4.0398e+04],
         [1.0000e+00, 7.4105e+04],
         [1.0000e+00, 7.6725e+04],
         [1.0000e+00, 1.8317e+04]]

y_fr = [[2503],
         [2992],
         [1042],
         [2487],
         [2014],
         [2805],
         [5062],
         [2643],
         [2126],
         [1784],
```

```
[2641],
[2766],
[5047],
[5312],
[1215]]
```

The results from this data were:

Language	Gradient Descent	SSE	Weights
English	Batch	0.00086294	(-3.56432855, 0.06430049)
English	Stochastic	0.00105848	(16.51596702, 0.06423801)
French	Batch	0.00100614	(8.76164958, 0.06830075)
French	Stochastic	0.00248123	(-13.79831984, 0.06761032)

3.2 Classification

The data used for the classification section consists of one normalized feature-label pair:

```
X = [[1.0, 35680.0, 2217.0],
     [1.0, 42514.0, 2761.0],
     [1.0, 15162.0, 990.0],
     [1.0, 35298.0, 2274.0],
     [1.0, 29800.0, 1865.0],
     [1.0, 40255.0, 2606.0],
     [1.0, 74532.0, 4805.0],
     [1.0, 37464.0, 2396.0],
     [1.0, 31030.0, 1993.0],
     [1.0, 24843.0, 1627.0],
     [1.0, 36172.0, 2375.0],
     [1.0, 39552.0, 2560.0],
     [1.0, 72545.0, 4597.0],
     [1.0, 75352.0, 4871.0],
     [1.0, 18031.0, 1119.0],
     [1.0, 36961.0, 2503.0],
     [1.0, 43621.0, 2992.0],
     [1.0, 15694.0, 1042.0],
     [1.0, 36231.0, 2487.0],
     [1.0, 29945.0, 2014.0],
     [1.0, 40588.0, 2805.0],
     [1.0, 75255.0, 5062.0],
     [1.0, 37709.0, 2643.0],
     [1.0, 30899.0, 2126.0],
     [1.0, 25486.0, 1784.0],
     [1.0, 37497.0, 2641.0],
     [1.0, 40398.0, 2766.0],
     [1.0, 74105.0, 5047.0],
     [1.0, 76725.0, 5312.0],
     [1.0, 18317.0, 1215.0]]

y = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```

```
0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

The results from this data were:

Classification Type	AV Accuracy	Weights
Perceptron	0.9666...	(0, -0.26348008, 3.98286898)
Logistic Regression	0.9666...	(2.72150342, -0.04914648, 0.74381126)

4 Optimization Algorithms

In this last section, the optimization algorithms for stochastic gradient descent (SGD) from Ruder's paper are summarized briefly.

4.1 Momentum

Momentum is a method which helps converge in ravines and damps oscillations. It does this by adding a portion of the previous update vector to the current, thus mimicking the accumulated momentum of a rolling ball. This results in faster convergence rates.

4.2 Nesterov Accelerated Gradient

Similarily to momentum, the Nestrov accelerated gradient also calculates the previous accumulated vector (momentum). It does this before calculating the current gradient, makes the leap, then calculates a new gradient at that approximate future point and corrects itself accordingly. This anticipatory update prevents the "ball" from rolling too fast and increases performance.

4.3 Adagrad

Adagrad is a mechanism which adapts the learning rate with respect to the weight parameters. At every point in time, for every weight parameter, the learning rate is adjusted (divided) with respect to the sum of the squares of the previous gradients. This eliminates the need to manually tune the learning rate but makes the learning rate decrease with time until the algorithm no longer makes progress towards an optimum.

4.4 Adadelta & RMSprop

Both Adadelta and RMSprop are extensions of Adagrad which solve the problem of decreasing learning rate. They do this by adjusting the learning rate with respect to the decaying average of all past squared gradients, e.g the root mean squared error of past gradients. This eliminates the problem of a growing denominator which "kills" the learning rate with time. In addition to this, in Adadelta the learning rate is removed all together and replaced by the root mean squared error of parameter updates. This yields an algorithm which adjusts the learning rate fully autonomously and does not need an initial value for the learning rate.

4.5 Adam

Adam also computes adaptive learning rates for each of the weight parameters. Besides the decaying average of the past squared gradients, Adam also keeps track of the decaying average of past gradients which is an estimate of the first moment. I'm not sure why this is a good solution but works well and compares favorably with other adaptive learning rate strategies according to the author.

4.6 AdaMax

AdaMax is a further improved version of Adam which solves the problem of bias towards zero caused by the initial zero vectors. As far as I understand it, this is done by dividing the learning rate with a maximum operation between the norm of previous gradients and the current.

4.7 Nadam

Nadam is a combination of the Adam and Nesterov Accelerated Gradient techniques. Just like in Nestov, the momentum step is taken first but here we also combine it with individual updates and adapted learning rates for each weight parameter.