Assignment 2 : Classification with the perceptron and logistic regression

Christoffer Lindell Bolin

February 21, 2022

1 Background

In this assignment, we were tasked to implement supervised machine learning with linear regression using gradient descent and linear classifiers using perceptron and logistic regression. The original french novel Salammbô and the translated version in english was used as the training data-sets. More specifically, the data-sets contained the letter counts in the 15 chapters and corresponding number of A's for the novel versions. The assignment was divided into two parts. In the first part of the assignment, we should be able to predict from either the french or english version the number of A's given the character counts. In the second part, we should be able to classify whether it is the french or english version based on the letter counts and corresponding A's. Since we have been dealing in this assignment with sparse data, as Ruder[1] suggest an adaptive learning-rate method could have been used instead of batch and stochastic gradient descent to improve performance.

2 Solution

Due to a limited knowledge with python, I took major inspiration from Pierre Nugues soulution[2] for the first part. The only major difference is regarding the normalization part, I did it a bit different to get a better understanding of how it works. As can be seen below. The full solution is avaible on Github[3]

```
X = np.array(X)
y = np.array([y]).T
maxima_X = np.amax(X, axis=0)
maxima_y = np.amax(y, axis=0)
maxima = np.concatenate((maxima_X, maxima_y))

X_max = max(X[:, 1])
X_min = min(X[:, 1])
```

[1]Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *Insight Centre for Data Analytics, NUI Galway* (2017). URL: https://arxiv.org/abs/1609.04747

[2]*Pnugues Github*. URL: https://github.com/pnugues/ilppp.

[3]*Github code*. URL: https://github.com/chrisse22222/AI_Course.

```
for i in range(len(X)):
    X[i][1] = (X[i][1] - X_min) /(X_max - X_min)

y = (y - np.min(y))/(np.max(y) - np.min(y))
```

Now to the fun part, classification. When it comes to Perceptron, a predict function was implemented that the batch descent function takes into consideration when calculation the loss. It was based on the slides from the course and returns the predicted class given the weight vector and the the current row in the X matrix ([Class, Characters, A's]). Where 1.0 represent the french and 0.0 the english version.

```
def predict(X_row, w):
    return 1.0 if w[0] + w[1] * X_row[1] + w[2] * X_row[2] >= 0.0 \
        else 0.0
```

I decided to use Stochastic batch descent for fitting the equation, mainly due to it being considered faster than the vanilla batch descent according to Ruder^1. It is quite similar to Pierre's implementation of stochastic decent, but you can notice that α is missing. This is due to the perceptron learning rule, where α is usually set to 1, hence it is not needed. The other major difference is an optimization that stops training when the number of misclassifed examples are high, also known as early stopping. This prevents overfitting the dataset, which reduces it accuracy against unseen data. Hence, defeating machine learnings purpose. See section examples for how this algorithm performed.

```
def fit_stoch(X, y, epochs=1000, max_misclassified=1000, verbose=True):
    w = [0.0, 0.0, 0.0]
    misclassified = 0
    index = list(range(len(X)))
    random.seed(0)

    for e in range(epochs):
        random.shuffle(index)
        for i in index:
            loss = y[i] - predict(X[i], w)
            if loss != 0:
                gradient = loss * np.array([X[i]]).T
                w = w + gradient.T[0]
                misclassified += 1
        if misclassified >= max_misclassified:
            break
    return w
```

Logistic regression works a bit different and here a logistic funtion was implemented. The major problem I had with this was the fact that I got an overflowerror, I could not seem to found a solution to it, so instead, I catch the exception. And I know for a fact, that if X is large > 0, it would be a case of division by 1 and the opposite for a smaller term < 0. It seems a bit sketchy, but it does the trick.

```
def logistic(x):
    try:
        return 1/(1 + math.exp(-x))
    except OverflowError:
        return 1 if x > 0 else 0
```

The next part is sort of straightforward, so I will not explain it in detail. In predict probability, a vector of probabilities belonging to class 1.0 (french) is returned. In the predict function, a matrix with the predicted classes is returned based on the probabilities from the previous function mentioned.

```
def predict_proba(X, w):
    P = []
    for i in range(len(X)):
        P.append(logistic(np.matmul(X[i], w)))

    return P

def predict(X, w):
    P = predict_proba(X, w)
    return [(1 if x >= 0.5 else 0) for x in P]
```

Here is my stochastic batch descent with the logistic regression implemented, it is quite similar to perceptron but the loss function is swapped.

```
def fit_stoch(X, y, alpha=100, epochs=1000, epsilon=1.0e-4,
 verbose=False):
    w = [0.0, 0.0, 0.0]
    index = list(range(len(X)))
    random.seed(0)

    for epoch in range(epochs):
        random.shuffle(index)
        for i in index:
            loss = y[i] - predict(X, w)[i]
            if loss != 0:
                gradient = loss * np.array([X[i]]).T
                w = w + alpha * gradient.T[0]

        if np.linalg.norm(gradient) < epsilon:
            break

    return w
```

2.1 Examples

Before we dive into the example outputs I get, I just quickly wanna mention how the performance of these two approaches of classification are evaluated. Which is with the leave-one-out cross validation method. In our case, 30 models had to be trained. In each of those sessions, out of the 30 samples of data we had, only 29 were used for training, while the remaining sample to evaluate the performance to see if it guessed correct. As can be seen below, is how I have implemented it.

```
def leave_one_out_cross_val(X, y, fitting_function):
    rounds = len(X) # 30 rounds
    score = 0

    for i in range(rounds):
        X_copy = X.copy()
        y_copy = y.copy()
        x_sel = X_copy.pop(i)
        y_sel = y_copy.pop(i)
        w = fitting_function(X_copy, y_copy)
        if y_sel - predict(x_sel, w) == 0:
            score += 1
    return float(score/rounds)
```

For the perceptrion part, the result of my weight vector are as follows with a cross-validation accuracy of ≈ 0.93.

w0	w1	w2
0.0	-3.8056305	3.98286898

Meanwhile, for the logistic regression part, the result are the following with a cross-validation accuracy of ≈ 0.97. This is a slight improvement from the perceptrion part.

w0	w1	w2
0.0	-380.56304985	398.28689759

3 Gradient descent optimization algorithms

Ruder mentions in his article[1] a few different approaches to enhance performance of mini-batch gradient descent, also known as SGD. SGD is sort of a hybrid solution between batch and stochastic gradient descent, it combines the best from both worlds. Despite this, he emphasize that SGD does not guarantee good convergence and that choosing a proper learning rate α can be difficult. He further describes that another major challenge SGD has is getting trapped in numerous suboptimal local minima due to saddle points where one dimension slopes up and another down. This had lead to a number of spin-of algorithms to tackle the main issues with SGD. Whereas the most promising seems to be Adagradm Adadelta and RMSprop that seemed to immediately head in the right direction and converged correspondingly as fast.

Adagrad is an adaptive learning-rate method. As the naming suggest, it adapts the learning rate to the parameters where it performs larger updates for infrequent and smaller for frequent parameters. Which makes it a good fit for dealing with sparse data. Adadelta is an extension of Adagrad which aims to reduce its predecessors aggressiveness by monotonically decreasing the learning rate. Meanwhile RMSprop is also an adaptive learning rate method which has been developed around the same time as Adadelta independently to tackle the Adagrad's diminishing learning rates and is comparable to Adadelta.

Despite this, further optimizations could be made. For example by normalizing batches enables a higher learning rates, implement early stopping to stop training if the validation error does not improve enough and shuffling the training examples to prevent any sort of bias for the optimization algorithm.

References

Github code. URL: `https://github.com/chrisse22222/AI_Course`.
Pnugues Github. URL: `https://github.com/pnugues/ilppp`.
Ruder, Sebastian. "An overview of gradient descent optimization algorithms". In: *Insight Centre for Data Analytics, NUI Galway* (2017). URL: `https://arxiv.org/abs/1609.04747`.