Linear regression in Python - Project report

Zeineb Rejiba

August 2020

1 Project description

This project aims at creating a Docker application that allows to train a linear regression model using *batch gradient descent*. Both univariate and multivariate regression types are supported. In the former case, the task consists in predicting profits for a food truck company based on the population size of a given city. In the latter case, the goal is to predict house prices based on two variables which are the size of the house (in square feet) and the number of bedrooms it has.

2 Tools

The project's implementation is based on Python, *Tensorflow*[1] and *scikit-learn*[2]. More specifically, Tensorflow is used for tensor operations such as multiplication and gradient calculations, whereas scikit-learn is used to provide feature scaling in the multivariate regression case. It is worth noting that scikit-learn does provide a *LinearRegression* object, which does linear regression out-of-the-box. However, it is done using the closed form solution and not based on batch gradient descent, as required in the first parts of this exercise. That is why, it has not been used.
All code is packaged in a Docker image that can automatically run the different steps of the exercise[3].

3 Project structure

Fig. 1 shows the structure of this project. More in detail, it is comprised of the following directories and files:

- data : contains the training data for univariate/multivariate regressions.
- data_utils.py : contains utility functions that facilitate the task of loading and preparing the data.
- Dockerfile : defines the base image to be used by the docker container, as well as the necessary commands to configure this image.
- figs : this directory will be filled with the figures that will be created as part of the exercise.
- lin_reg.py : this is the main file. It runs the different steps of the exercise, as will be explained later.
- model.py : contains the definition of the class corresponding to the linear regression model. It is inspired by Tensorflow's tutorial on custom training[4] and generalizes it to the multivariate case. It is characterized with a weight vector θ and methods allowing to update those weights and using them to make predictions.
- plot_utils.py : contains utility functions to create the different plots in each step of the exercise.
- README.md: contains the project's description and instructions on how to run it.

[1] https://www.tensorflow.org/
[2] https://scikit-learn.org/stable/
[3] Note that in a previous version of this code, I created a CLI application that executes specific steps of the exercise using a specified argument. However, to simplify things, I later removed this to a version where the different steps are executed sequentially, without the need for user input.
[4] https://www.tensorflow.org/tutorials/customization/custom_training

```
lin-regression
├── data/
│   ├── ex1data1.txt
│   └── ex1data2.txt
├── data_utils.py
├── Dockerfile
├── figs/
│   └── .gitignore.txt
├── lin_reg.py
├── model.py
├── plot_utils.py
└── README.md
```

Figure 1: Project structure

4 Implementation steps

In the following, the different steps of the exercise will be described. Such steps are executed in sequence in the *lin_reg.py* main file, which will make use of the Model class as well as utilities defined in *data_utils.py* and *plot_utils.py*.

4.1 Linear regression with one variable (Step 2)

This step will use the data in *ex1data1.txt* to predict profits based on the population size variable.

4.1.1 Plotting the Data (Step 2.1)

The first step is to visualize the training data to see how it is distributed and identify any potential shapes it may contain. The result is shown in Fig. 2.

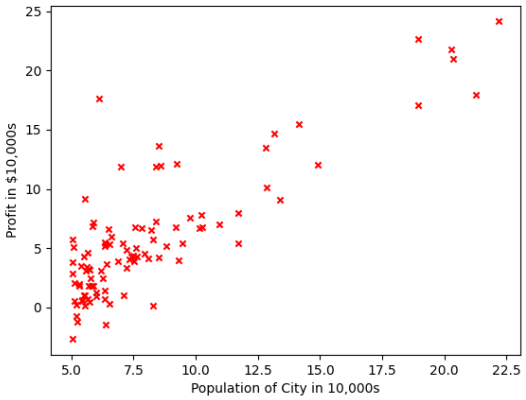

Figure 2: Scatter plot of training data

The idea is to find the weights/coefficients θ_0 and θ_1 such that the line with the equation $h_\theta(x) = \theta_0 + \theta_1 x$ constitutes a best fit for the training data points.

4.1.2 Gradient descent (Step 2.2)

Finding the best weights can be done using the gradient descent algorithm. Its main idea is to start with random weight values, make predictions using those weight values and calculate the mean squared error between the predicted values and the true target values.

Then, the weights are adjusted using the calculated error and a learning rate α that controls the learning speed as follows:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)}, \; j \in 0, 1 \tag{1}$$

This process is repeated iteratively, until convergence or for a specified number of iterations. In batch gradient descent, each iteration adjusts the weights using the *whole* training data (of size m).

The part corresponding to Step 2.2 in *lin_reg.py* is shown below. After preparing the data, the model will be initialized and then trained for 1500 iterations with a learning rate 0.01. Then, predictions for population sizes 35,000 and 70,000 will be performed:

```
X = X.to_numpy()  # convert dataframe to numpy array

X, y = prepare_data(X, y)

# initialize model using the new number of columns (number of features + column
    of 1s)
model = Model(X.shape[1])

# train model
nb_iterations = 1500
lr = 0.01
print('Training model using {:d} iterations and learning rate= {:.2f}'.format(
    nb_iterations, lr))
model.train(nb_iterations, X, y, learning_rate=lr)

#make predictions using the provided examples: population sizes of 35,000 and
    70,000
ex_x1 = tf.constant([1.0, 3.5], shape=(1, 2))
ex_y1 = model.predict(ex_x1).numpy()[0][0]

print('When the size of the population is 35,000 => The predicted profit is {:.2
    f}$'.format(
    ex_y1 * 10000))

ex_x2 = tf.constant([1.0, 7.0], shape=(1, 2))
ex_y2 = model.predict(ex_x2).numpy()[0][0]

print('When the size of the population is 70,000 => The predicted profit is {:.2
    f}$'.format(
    ex_y2 * 10000))
```

The previous code makes use of the `train` method defined in *model.py*. This method will train the linear regression model for the specified number of iterations.

```
def train(self, nb_iter, X, y, learning_rate):
    """
    Trains the linear model for the specified number of iterations
    :param nb_iter: number of iterations used for training
    :param X: an m by (n+1) matrix containing the inputs, where m is the number
        of rows
    :param y: the target values
    :param learning_rate: the learning rate
    """
    for i in range(nb_iter):
        self.update(X, y, learning_rate)
    print('Training done!')
```

In each iteration of the loop, the whole training data will be used by the `update` method of the `Model` to perform weight updates, based on the provided learning rate. This process involves the following steps:

- Loss/cost calculation: This is done via the `calculate_loss` method.
- Gradient calculation: It makes use of Tensorflow's `tf.GradientTape` to record the operations being performed and the `.gradient` method to calculate the derivative of the weight vector `self.theta`.
- Weight updating: It is done using the `.assignsub` method that combines assignment and subtraction in a single method.

```
    def update(self, inputs, outputs, learning_rate):
        """
        updates the weights by performing gradient descient on the entire batch of
            inputs
        :param inputs: an m by (n+1) matrix containing the inputs, where m is the
            number of rows
        :param outputs: the target values
        :param learning_rate: the learning rate
        """
        with tf.GradientTape() as t:
            y_hat = self.predict(inputs)
            current_loss = self.calculate_loss(outputs, y_hat)

        # calculate the gradient with respect to the weight vector theta
        d_loss_d_theta = t.gradient(current_loss, self.theta)

        # update the weights theta_j = theta_j - learning_rate * d_loss_d_theta
        self.theta.assign_sub(learning_rate * d_loss_d_theta)  #  assign_sub
            combines tf.assign and tf.sub
```

The `calculate_loss` method used previously calculates the following equation:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2 \tag{2}$$

More specifically, it calculates the mean squared error by combining `tf.reduce_mean` and `tf.square` and multiplying the result by .5, as shown below:

```
    def calculate_loss(self, target_y, predicted_y):
        """
        calculates the mean squared loss between the target values and the predicted
             values
        :param target_y: the target values
        :param predicted_y: the predicted values
        :return: the mean squared loss
        """
        loss = (tf.reduce_mean(tf.square(target_y - predicted_y))) * 0.5
        self.losses.append(loss.numpy())
        return loss
```

As for predictions they can be made based on the equation $h_\theta(x) = \theta^T x$ using Tensorflow's `tf.matmul`, as shown in the next snippet.

```
    def predict(self, x):
        """
        performs a prediction for the given input, using the current weight values
        :param x: an m by (n+1) matrix containing the inputs, where m is the number
            of rows
```

```
        :return: predicted value
        """
        y_hat = tf.matmul(x, self.theta)
        return y_hat
```

4.1.3 Debugging (Step 2.3)

In order to show the training data along with the regression line for debugging purposes, the method `plot_data_with_line(df, model.theta)` is called in *lin_reg.py* and the result is shown in Fig. 3.

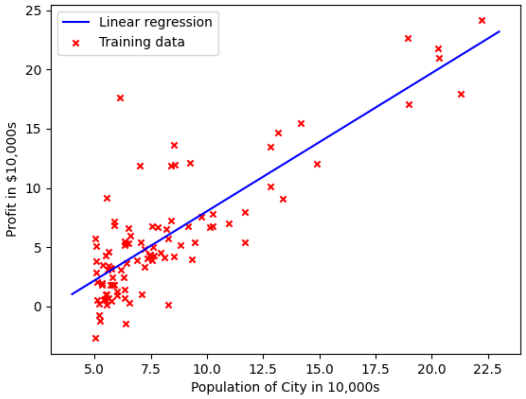

Figure 3: Training data with linear regression fit

4.1.4 Visualizing $J(\theta)$ (Step 2.4)

In order to plot the surface and the contour plot, the `plot_surface_and_contour(w, X, y)` method is called. The resulting plots are depicted in Fig. 4.

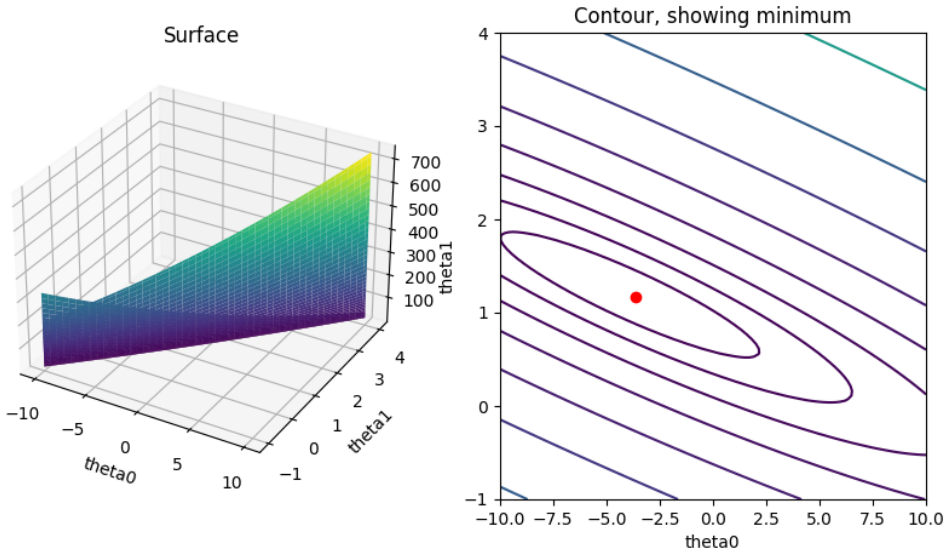

Figure 4: Cost function $J(\theta)$

4.2 Linear regression with multiple variables (Step 3)

This part deals with the case of linear regression using multiple variables, also known as multivariate regression. In the example used in the exercise, the goal is to predict house prices based on the area of the house and the number of rooms. The data can be found in the file *ex1data2.txt*

4.2.1 Feature normalization (Step 3.1)

Since the features considered in this example are on different scales (house areas are in the order of thousands whereas the number of rooms is in the range [1,5]), it is necessary to re-scale the inputs before the training process. This process is also called *feature normalization* and it consists in subtracting the mean of the values of each column from the old value of that column, and dividing the result by the standard deviation of that column. *scikit-learn* can do this by performing the import `from sklearn import preprocessing` and running the following code:

```
    scaler = preprocessing.StandardScaler().fit(X_multi) # this same scaler will be
        used later when making predictions
    X = scaler.transform(X_multi)
```

4.2.2 Gradient descent (Step 3.2)

As our implementation of the `Model` class is generic enough, no changes are needed to perform gradient descent in the multivariate case. So, it just amounts to the following lines:

```
X, y = prepare_data(X, y)

# initialize model
model = Model(X.shape[1])
lr = 0.01
nb_iterations = 500
print('Training model using {:d} iterations and learning rate= {:.2f}'.format(
    nb_iterations, lr))
model.train(nb_iterations, X, y, learning_rate=lr)
```

1. Selecting learning rates (Step 3.2.1):

   In the previous step, the learning rate selection was arbitrary. In this part, we investigate the impact of the learning rate on the convergence of gradient descent. As it can be seen from Fig. 5, when the learning rate was 0.3, gradient descent converged faster (blue line).

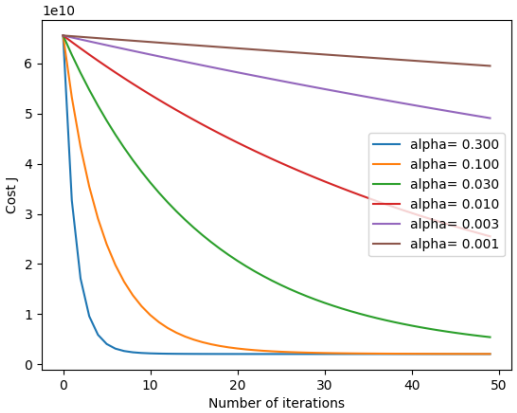

Figure 5: Performance of gradient descent under different learning rates

As a result, $\alpha = 0.3$ was used to perform the final training. The resulting weights were used to make predictions for the provided example. It is worth noting that while making predictions, it is necessary to re-scale the inputs using the same scales that have been used in the training process.

The part corresponding to this step in the code is highlighted below:

```
plot_cost_per_lr(X, y)

# from the figure, it looks like lr=0.3 is the best one. So, training will be
    performed using this value.
model = Model(X.shape[1])
lr = 0.3
nb_iterations = 500
print('Training model using {:d} iterations and learning rate= {:.2f}'.format(
    nb_iterations, lr))
model.train(nb_iterations, X, y, learning_rate=lr)

# when the regression is with multiple variables we need to take into account the
    feature scaling done previously when predicting on new values
# use the scaler to apply the same transformation as the training data
x_multi_scaled = scaler.transform(np.array([1650, 3]).reshape(1, -1))

# add the 1s for the intercept and create a tensorflow constant for the test example
ex_x_multi = tf.constant(np.concatenate([np.ones((1, 1)), x_multi_scaled], axis=1),
    shape=(1, 3), dtype=tf.float32)
ex_y_multi = model.predict(ex_x_multi).numpy()[0][0]

print(
    'When the house has an area of 1650 square feet and 3 bedrooms => Its predicted
        price is {:.2f}\$'.format(
        ex_y_multi))
```

4.2.3 Normal equations (Step 3.3)

Besides using gradient descent, another approach to find the optimal weights for linear regression is by using the closed form solution provided by the normal equations:

$$\theta = \left(X^T X\right)^{-1} X^T \vec{y} \tag{3}$$

Eq. 3 can be implemented in Python as follows, where `@` is the multiplication operator and `.T` is the transpose operator, whereas `np.linalg.inv()` performs matrix inversion:

```
theta_neq = np.linalg.inv(X.T @ X) @ X.T @ y
```

If the instructions in the README.md have been followed correctly, the output should look as shown in Fig. 6. As it can be seen, the solution obtained by gradient descent is very close to the one obtained by the normal equations. You may also note that the directory *figs* will be populated with the different plots created during execution.

```
C:\Users\Zeineb\linear-regression>docker run -v %cd%\figs:/lin_reg/figs --name regression regression-
Step 2: Linear regression with one variable
Step 2.1 Plotting the data
Saving plot to figs/scatter_plot_training_data.png


Step 2.2 Gradient desent
Training model using 1500 iterations and learning rate= 0.01
Training done!
When the size of the population is 35,000 => The predicted profit is 4519.76$
When the size of the population is 70,000 => The predicted profit is 45342.45$


Step 2.3: Debugging
Slope=  1.1663625
Intercept=  -3.6302927
Saving plot to figs/data_with_line.png


Step 2.4: Visualizing J(theta)
Saving plot to figs/surface_and_contour.png


Step 3: Linear regression with multiple variables
Step 3.1 Feature normalization


Step 3.2 Gradient descent
Training model using 500 iterations and learning rate= 0.01
Training done!


Step 3.2.1 Selecting learning rates
Training model using  50 iterations and learning rate= 0.3
Training done!
Training model using  50 iterations and learning rate= 0.1
Training done!
Training model using  50 iterations and learning rate= 0.03
Training done!
Training model using  50 iterations and learning rate= 0.01
Training done!
Training model using  50 iterations and learning rate= 0.003
Training done!
Training model using  50 iterations and learning rate= 0.001
Training done!
Training model using 500 iterations and learning rate= 0.30
Training done!
When the house has an area of 1650 square feet and 3 bedrooms => Its predicted price is 293081.44$


Step 3.3 Normal equations
When the house has an area of 1650 square feet and 3 bedrooms => Its predicted price is 293081.47$
```

Figure 6: Execution output