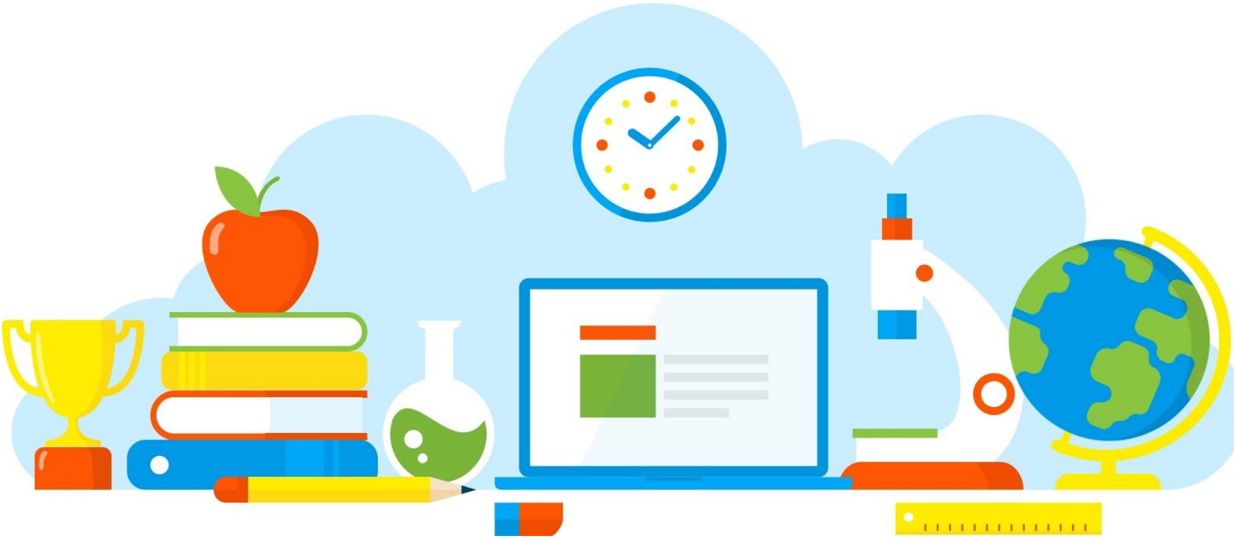


Machine Learning HW4

Teacher: Dr.Zare

Mahdi Sadeghi
830598035

Loading and Processing Data

There are three functions that process dataset in different ways: standardising, log(x+0.1) and binarization. These functions attach an all one column to dataset too.

```
def load_dataset(file_name, t):
    """
    loads dataset and returns it
    :param file_name:
    :param t: type of processing on dataset (s: standard, l: log, b:
binary)
    :return:
    """
    file_location = path.join(path.abspath(path.dirname(__file__)),
'../data', file_name)
    dataset = np.loadtxt(file_location, dtype=np.float128)

    if t == 's':
        return standardize_dataset(dataset)
    elif t == 'l':
        return logplus_dataset(dataset)
    elif t == 'b':
        return binarize_dataset(dataset)


def standardize_dataset(dataset):
    """
    standardizes a dataset
    :param dataset:
    :return:
    """
    x = dataset[:, :-1]
    dataset[:, :-1] = (x - x.mean(axis=0)) / np.std(x, axis=0)
    return dataset


def logplus_dataset(dataset):
    """
    computes log(a + 0.1) on all elements of input
```

```
    :param dataset:
    :return:
    """
    x = dataset[:, :-1]
    dataset[:, :-1] = np.log(x + 0.1)
    return dataset


def binarize_dataset(dataset):
    """
    if a > 0 => that element will become 1 else => 0
    :param dataset:
    :return:
    """
    x = dataset[:, :-1]
    binary_x = x.copy()
    binary_x[binary_x > 0] = 1
    binary_x[binary_x <= 0] = 0
    dataset[:, :-1] = binary_x
    return dataset


def generate_training_test_datasets(dataset):
    """
    splites data to training and test datasets
    :param dataset:
    :return: training, test
    """
    np.random.shuffle(dataset)
    split_boundary = math.floor(80 * dataset.shape[0] / 100)
    training_dataset, test_dataset = dataset[:split_boundary],
dataset[split_boundary:]
    return training_dataset, test_dataset
```

Gradient Descent

Gradient descent is used to calculate coefficients of the model. In each iteration the cost difference is checked to see if it is less than a threshold. If it is, the iterations are stopped. All the functions work with matrices:

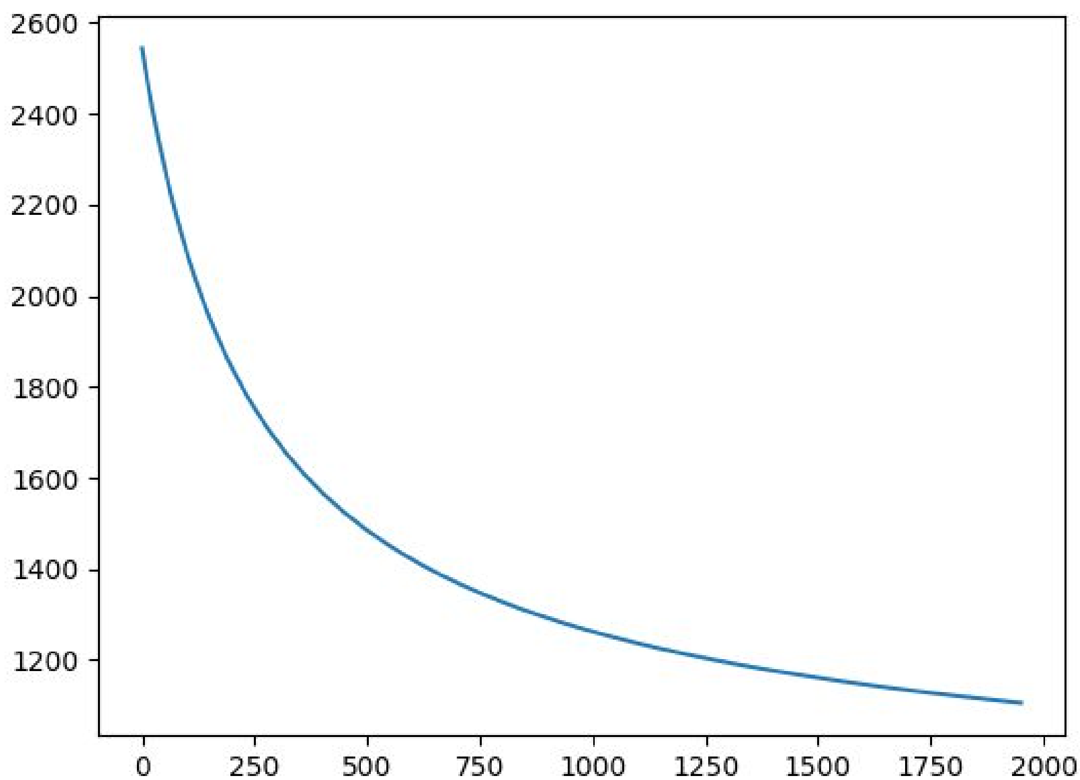
```
def compute_descent_size(x, y, beta):
   """
   computes the amount of descent for each parameter
   :param x: i*j matrix
   :param y: i*1 vector
   :param beta: j*1 vector
   :return: j*1 vector
   """
   return ((h(x, beta) - y).T.dot(x)).T


def gradient_descent_step(x, y, beta):
   """
   discends beta parameters for a single step and returns the result
   :param x: i*j matrix
   :param y: i*1 vector
   :param beta: j*1 vector
   :return: j*1 vector (new beta values)
   """

   return beta - (ALPHA * compute_descent_size(x, y, beta))

def gradient_descent(training_dataset):
   """
   computes beta using gradient descent algorithm and computes list
of costs
   :param training_dataset:
   :return: tuple containing beta (j*1 vector), costs list (list of
costs in each iteration)
   """
```

```
# attach a column of 1s to the beginning of x
x = training_dataset[:, :-1]
x = np.hstack((np.matrix(np.ones(x.shape[0])).T, x))

# save targets in separate variables
y = training_dataset[:, -1].reshape((x.shape[0], 1))

beta = np.zeros((x.shape[1], 1))

costs_list = []
last_cost = math.inf
current_cost = 0

while abs(last_cost - current_cost) > THRESHOLD:
    beta = gradient_descent_step(x, y, beta)

    last_cost = current_cost
    current_cost = cost(x, y, beta)

    costs_list.append(current_cost)

return beta, costs_list
```

Results

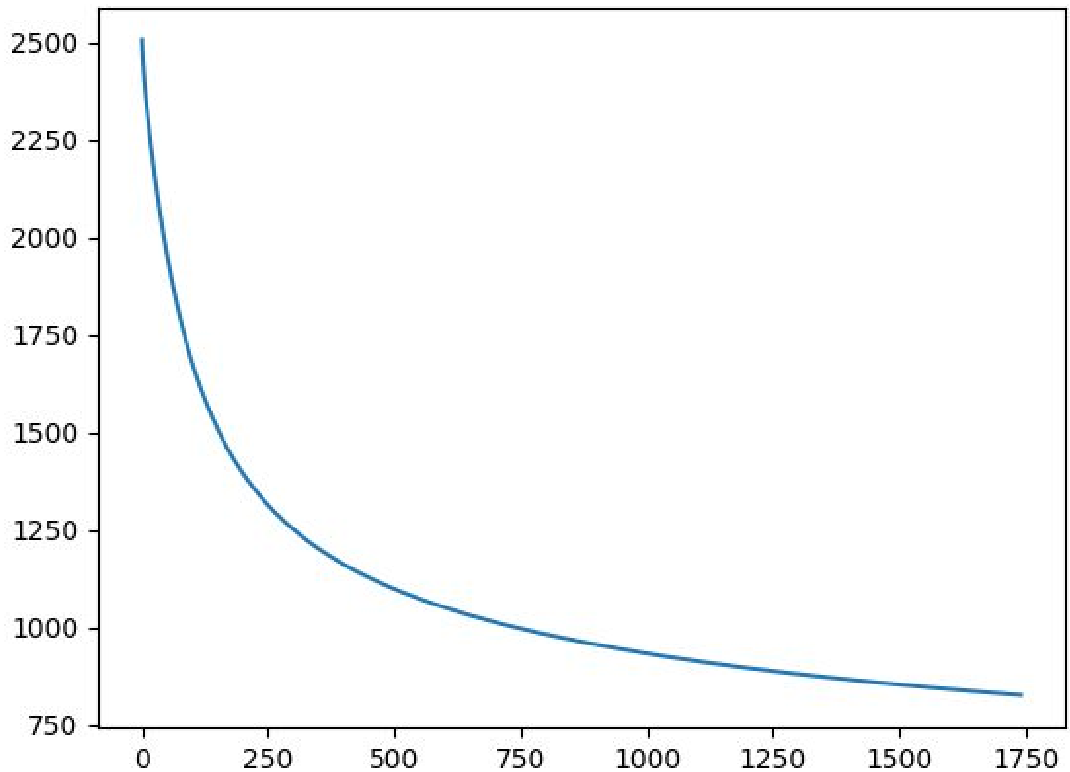
These results were obtained with probability threshold of 0.5:

Standardized Dataset:



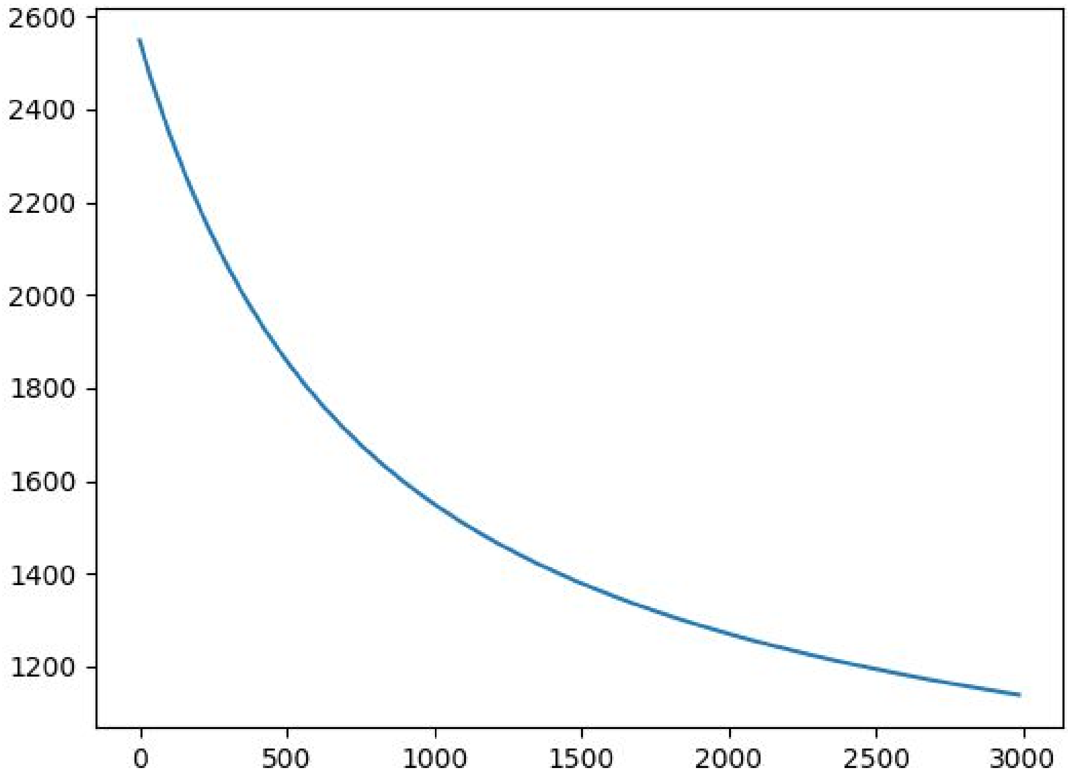
Error Rate: 0.6134636264929425

Log(x + 0.1) Dataset:



Error Rate: 0.6232356134636265

Binary Dataset:



Error Rate: 0.5776330076004343