HW5_P1

March 5, 2022

1 Problem 1

The LinearRegressor class encapsulates all modules that're required for training. The Homework5P1 class works on all the subproblems and outputs relevant figures and numbers.

[20]:
```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

np.random.seed(0) # to produce same random numbers

ROOTDIR = './data/h5w7_pr1_dataset_files/'
TRAIN_FILE = 'h5w7_pr1_power_train.csv'
TEST_FILE = 'h5w7_pr1_power_test.csv'


class LinearRegressor:

    def __init__(self, n_iters = 100):
        self.n_iters = n_iters


    def _init_weights(self, dims):
        """
        Initialize weights with random values from a uniform distribution [-0.
↪1, 0.1]

        Parameters
        ----------
        dims : int
            number of dimensions

        Returns
        -------
        np.array
            weights
        """
```

```python
        # (a)(ii)
        # init weight of shape (1, dims = 5) (augmented space)
        return np.random.uniform(-0.1, 0.1, dims)


    def _predict(self, x, weights):
        """
        Predict the output of the model (w.transpose * x)

        Parameters
        ----------
        x : np.array
            input data
        weights : np.array
            weights

        Returns
        -------
        float
            predicted output
        """
        # w.transpose * x
        return np.dot(weights, x).astype(float)


    @staticmethod
    def _shuffle_data(x, y):
        """
        Randomly shuffle the data

        Parameters
        ----------
        x : np.array
            input data
        y : np.array
            output data

        Returns
        -------
        np.array, np.array
            shuffled input and output data
        """
        assert len(x) == len(y), "unequal number of data points."
        p = np.random.permutation(len(x))
        return x[p], y[p]
```

```python
    @staticmethod
    def _lr_schedular():
        """
        Return combinations of learning rates (A, B)

        Parameters
        ----------
        None

        Returns
        -------
        list
            list of tuples of learning rates (A,B)
        """
        lr_combo = []
        for a in [0.01, 0.1, 1, 10, 100]:
            for b in [1, 10, 100, 1000]:
                # combo = (a, b)
                lr_combo.append((a, b))
        return lr_combo


    def _init_rms(self, x, y, init_weights):
        """
        Compute initial E[RMS] on the entire trianing set by using the initial␣
↪weights.t

        Parameters
        ----------
        x : np.array
            input data
        y : np.array
            output data
        init_weights : np.array
            initial weights

        Returns
        -------
        float
            initial E[RMS]
        """
        J_w = 0.0
        for idx in range(len(x)):
            J_w += (self._predict(x[idx], init_weights) - y[idx]) ** 2
        return (J_w / len(x)) ** 0.5
```

```
    def _fit(self, x, y):
        """
        Linear MSE regressor by using sequential gradient descent. Fit the␣
↪model to the training set.

        Parameters
        ----------
        x : np.array
            input data
        y : np.array
            output data

        Returns
        -------
        lr_rms : list
            list of tuples of learning rates (A,B), corresponding RMS, and␣
↪epoch number.
        weight_store : list
            list of weights at each epoch
        """
        lr_rms = []
        weight_store = []
        x, y = self._shuffle_data(x, y)
        for (A, B) in self._lr_schedular():
                iters = 1
                converged = False
                print(f"A = {A}, B = {B}")
                rms = 0
                # for each (a,b), initialize weights, and compute initial RMS
                weights = self._init_weights(x.shape[1])
                init_rmse = self._init_rms(x, y, weights)
                while iters <= self.n_iters and not converged:
                    J_w = 0.0
                    # compute J_w for each data point
                    for idx in range(len(x)):
                        op = self._predict(x[idx], weights)
                        J_w += (op - y[idx]) ** 2 # compute J_w
                        lr = A / (B + iters) # learning rate
                        weights += -lr * ((op - y[idx]) * x[idx]) # update␣
↪weights
                    rms = (J_w / len(x)) ** 0.5 # compute RMS (after mean of␣
↪J_w)
                    lr_rms.append((A, B, rms, iters)) # storing (A, B, RMS,␣
↪iters)
                    weight_store.append(weights) # storing weights at each epoch
                    # halting conditions
                    if rms < 0.001 * init_rmse:
```

```
                    print(f'iv.1. RMSE is less than 0.001 of init_rmse at␣
↪iteration : {iters}')
                    converged = True
                    break
                if iters == 100:
                    print('iv.2. 100 epochs have been completed.')
                    converged = True
                    break
                iters += 1
        return lr_rms, weight_store


    def _min_rms_locations(self, lr_rms):
        """
        Find the locations of the minimum RMS (minimum of the tuple (A,B))

        Parameters
        ----------
        lr_rms : list
            list of tuples of learning rates (A,B), corresponding RMS, and␣
↪epoch number.

        Returns
        -------
        locs_rms_pairs : list
            list of locations of minimum RMS for each pair of (A, B).
        """
        idx = 0
        # extract rms from each tuple (A,B)
        rms = [row[2] for row in lr_rms]
        # to store locations of minimum RMSE = final RMSE for each pair (A,B)
        locs_rms_pairs = []
        while idx < len(lr_rms):
            locs_rms_pairs.append(rms.index(min(rms[idx : idx + self.n_iters])))
            # collect locations after every 100 iterations
            idx += self.n_iters
        return locs_rms_pairs
```

[21]:
```
class Homework5P1:
    def __init__(self,
                train_path,
                test_path,
                ):
        self.train_path = train_path
        self.test_path = test_path
```

```python
    def _get_features(self, path):
        """
        Get features from the data set.

        Parameters
        ----------
        path : str
            path to the data set

        Returns
        -------
        x : np.array
            features
        y : np.array
            labels
        """
        df = pd.read_csv(path, header = None)[1 : ]
        x, y = df.iloc[:, :-1].values.astype(float),\
               df.iloc[:, -1].values.astype(float)
        bias = np.ones((len(x), 1))
        x = np.concatenate((bias, x), axis = 1)
        return x, y


    def _load(self):
        """
        Load the data sets (train and test).

        Parameters
        ----------
        None

        Returns
        -------
        x_train : np.array
            features of training set
        y_train : np.array
            labels of training set
        x_test : np.array
            features of test set
        y_test : np.array
            labels of test set
        """
        self.train_x, self.train_y = self._get_features(self.train_path)
        self.test_x, self.test_y = self._get_features(self.test_path)
        return self.train_x, self.train_y, self.test_x, self.test_y
```

```
    def _plot_rms_epoch(self, lr_rms):
        """
        Obtain 5 plots for each pair of (A,B) and plot the RMS vs epoch.

        Parameters
        ----------
        lr_rms : list
            list of tuples of learning rates (A,B), corresponding RMS, and␣
↪epoch number.

        Returns
        -------
        None
        """
        idx = 0
        # extract list of rmse values and iterations (epochs here).
        rms, iters = [row[2] for row in lr_rms], [row[3] for row in lr_rms]
        while idx < len(lr_rms):
            plt.figure()
            # list of all rms values and iteration (epochs here) numbers for␣
↪each A.
            iters_a, rms_a = iters[idx : idx + 400], rms[idx : idx + 400]
            jdx = 0
            # plot for each pair of (A,B)
            while jdx < len(iters_a):
                plt.plot(iters_a[jdx: jdx + 100], rms_a[jdx: jdx + 100], '-',␣
↪label = lr_rms[jdx][1])
                plt.legend()
                jdx += 100
            plt.xlabel('Iterations')
            plt.ylabel('E[RMS]')
            plt.tick_params(axis = "x")
            plt.tick_params(axis = "y")
            plt.title(f'A = {lr_rms[idx][0]}, B (all values)')
            plt.savefig(f"A:{lr_rms[idx][0]}.png")
            idx += 400
        plt.show()


    def _error(self, y, y_hat):
        """
        Compute the RMSE error between the predicted and actual values.

        Parameters
        ----------
        y : np.array
```

```
            actual values
        y_hat : np.array
            predicted values

        Returns
        -------
        error : float
            RMSE error between the predicted and actual values
        """
        return (np.mean((y - y_hat) ** 2))** 0.5


    def _runner(self):
        """
        Runner module for the class.
        """
        # load data sets
        self.train_x, self.train_y, self.test_x, self.test_y = self._load()

        print("*** Fitting the model on the training data ***")
        model = LinearRegressor()
        # train model on training set
        lr_rms, weights = model._fit(self.train_x, self.train_y)

        print("*** Plotting RMS vs epochs for each pair of (A,B) ***")
        # plot RMS vs epoch (b)
        self._plot_rms_epoch(lr_rms)
        # find locations of best RMSE for each pair (A,B)
        best_rms_locs_pairwise = model._min_rms_locations(lr_rms)

        print("*** Printing RMSE values for each pair of (A, B) on the test set␣
↪***")
        # all possible combinations of (A,B)
        lr_combos = model._lr_schedular()
        point = 0
        # (d) RMS for each pair (A,B), on the test set
        for loc in best_rms_locs_pairwise:
            # compute predictions for best optimal weights for each pair (A,B)
            test_preds = [model._predict(weights[loc], self.test_x[idx]) for␣
↪idx in range(len(self.test_x))]
            rms = self._error(self.test_y, test_preds)
            print(f'E[RMS] is {round(rms, 4)} for A, B = {lr_combos[point]}')
            point +=1

        print("*** Trivial regressor *** ")
        # trivial regressor (e)
        trivial_y = np.array([np.mean(self.train_y)] * len(self.test_y))
```

```
        rms_trivial = self._error(self.test_y, trivial_y)
        print(f'E[RMS] for the trivial model is {round(rms_trivial, 4)}')
```

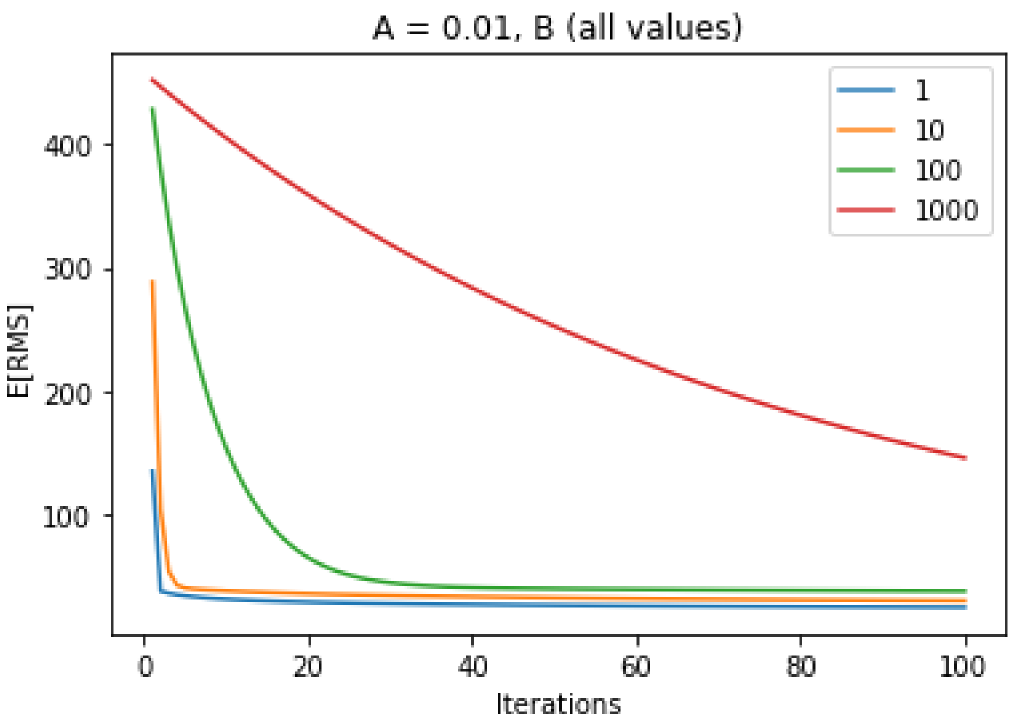
[23]:
```
if  __name__ == '__main__':
    train_file = os.path.join(ROOTDIR, TRAIN_FILE)
    test_file = os.path.join(ROOTDIR, TEST_FILE)
    hw = Homework5P1(train_file, test_file)
    hw._runner()
```

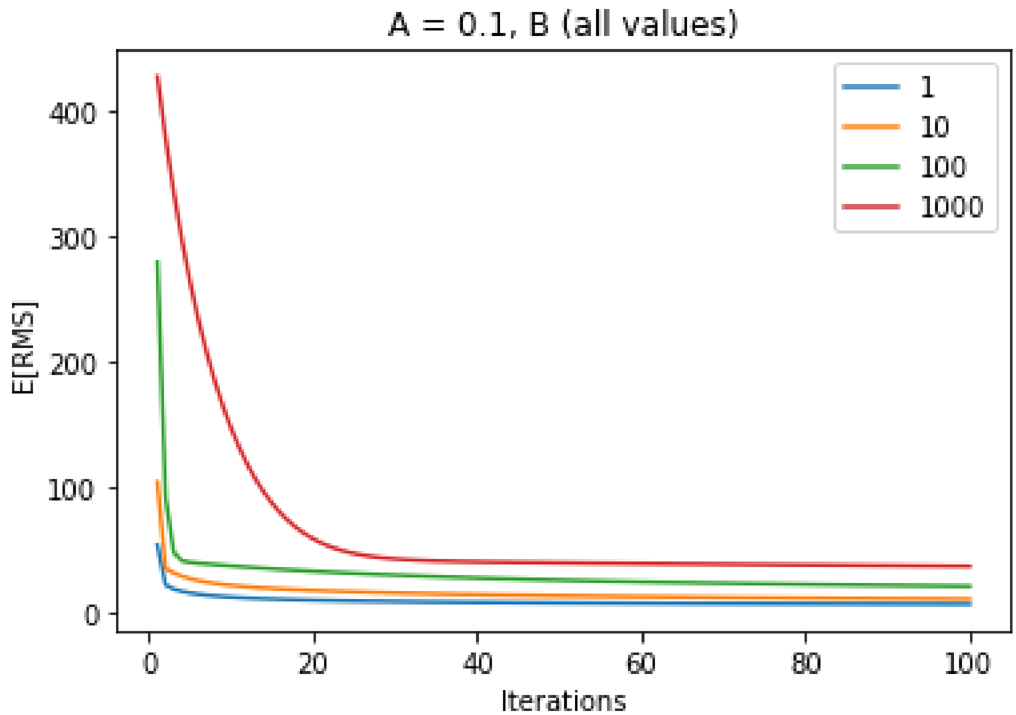
```
*** Fitting the model on the training data ***
A = 0.01, B = 1
iv.2. 100 epochs have been completed.
A = 0.01, B = 10
iv.2. 100 epochs have been completed.
A = 0.01, B = 100
iv.2. 100 epochs have been completed.
A = 0.01, B = 1000
iv.2. 100 epochs have been completed.
A = 0.1, B = 1
iv.2. 100 epochs have been completed.
A = 0.1, B = 10
iv.2. 100 epochs have been completed.
A = 0.1, B = 100
iv.2. 100 epochs have been completed.
A = 0.1, B = 1000
iv.2. 100 epochs have been completed.
A = 1, B = 1
iv.2. 100 epochs have been completed.
A = 1, B = 10
iv.2. 100 epochs have been completed.
A = 1, B = 100
iv.2. 100 epochs have been completed.
A = 1, B = 1000
iv.2. 100 epochs have been completed.
A = 10, B = 1
```

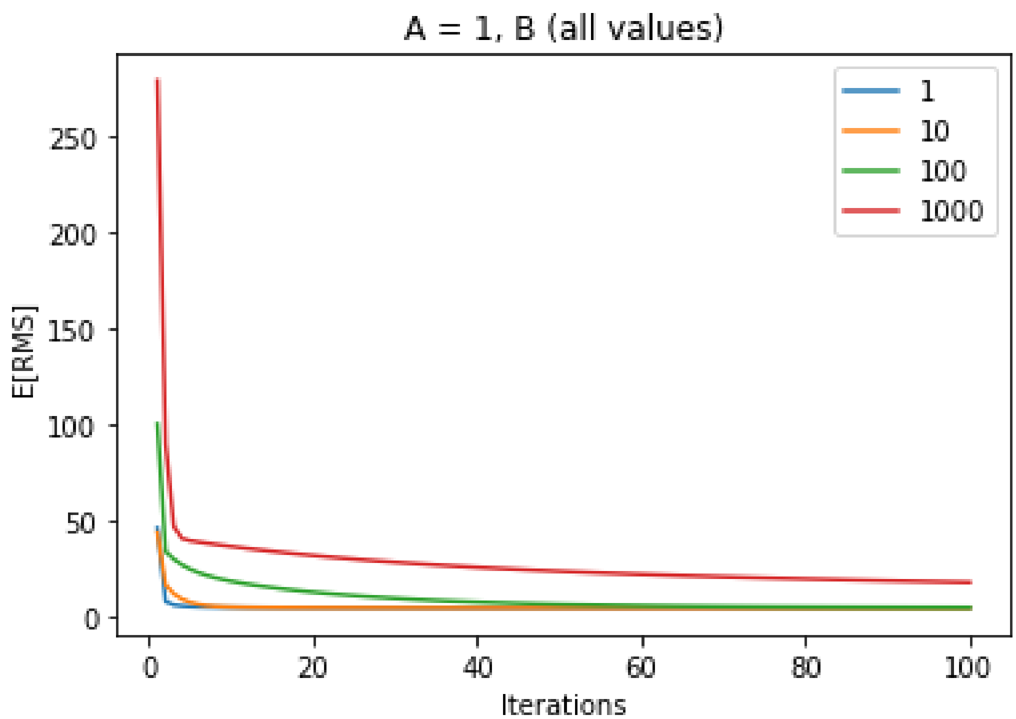
```
<ipython-input-20-76a7d2be38db>:162: RuntimeWarning: overflow encountered in
double_scalars
  J_w += (op - y[idx]) ** 2 # compute J_w
<ipython-input-20-76a7d2be38db>:164: RuntimeWarning: invalid value encountered
in add
  weights += -lr * ((op - y[idx]) * x[idx]) # update weights
```

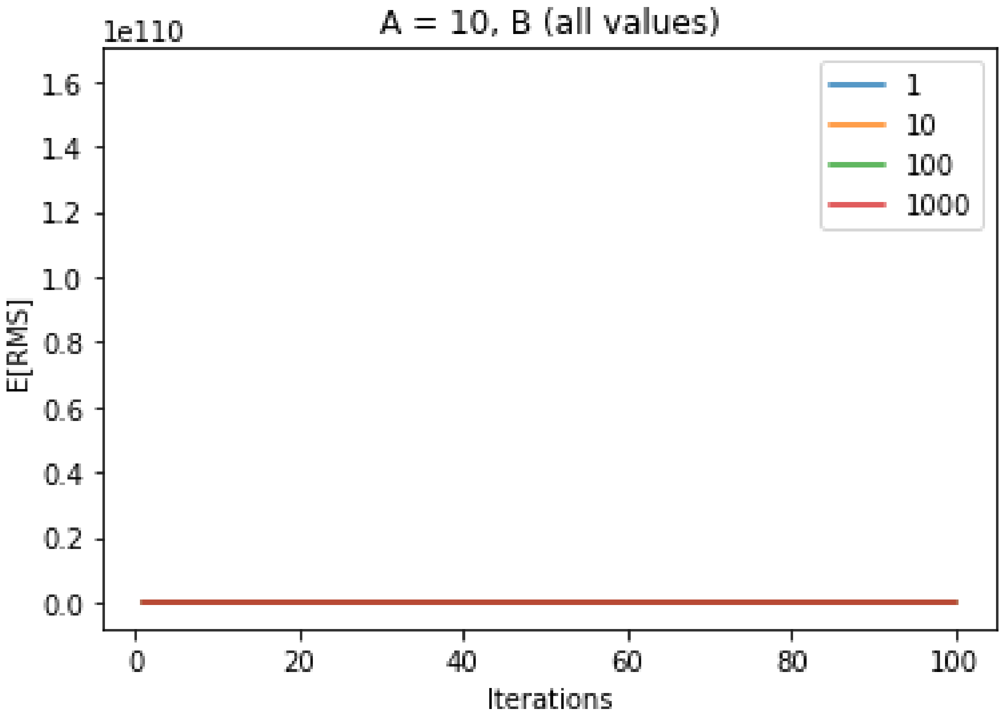
```
iv.2. 100 epochs have been completed.
A = 10, B = 10
iv.2. 100 epochs have been completed.
A = 10, B = 100
```

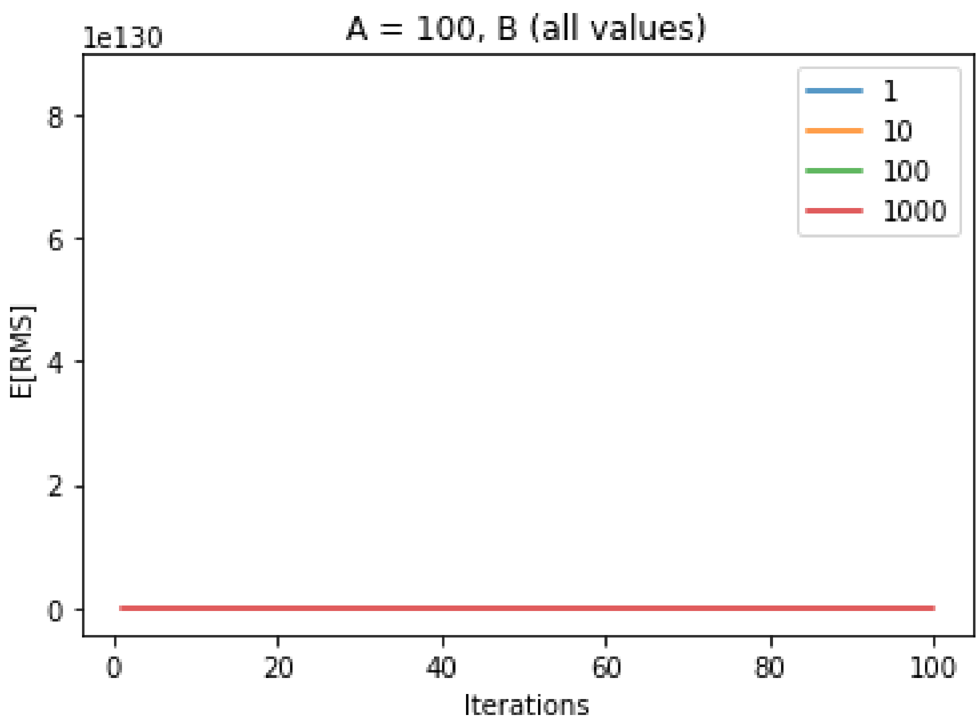
```
iv.2. 100 epochs have been completed.
A = 10, B = 1000
iv.2. 100 epochs have been completed.
A = 100, B = 1

<ipython-input-20-76a7d2be38db>:164: RuntimeWarning: overflow encountered in
multiply
  weights += -lr * ((op - y[idx]) * x[idx]) # update weights

iv.2. 100 epochs have been completed.
A = 100, B = 10
iv.2. 100 epochs have been completed.
A = 100, B = 100
iv.2. 100 epochs have been completed.
A = 100, B = 1000
iv.2. 100 epochs have been completed.
*** Plotting RMS vs epochs for each pair of (A,B) ***
```




A = 0.1, B (all values)
1
10
100
1000
E[RMS]
Iterations


A = 1, B (all values)
1
10
100
1000
E[RMS]
Iterations


A = 10, B (all values)
1e110
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
E[RMS]
0
20
40
60
80
100
Iterations
1
10
100
1000


A = 100, B (all values)
1e130
8
6
4
2
0
E[RMS]
0
20
40
60
80
100
Iterations
1
10
100
1000

```
*** Printing RMSE values for each pair of (A, B) on the test set ***
E[RMS] is 26.6521 for A, B = (0.01, 1)
E[RMS] is 33.0999 for A, B = (0.01, 10)
E[RMS] is 42.321 for A, B = (0.01, 100)
E[RMS] is 148.8757 for A, B = (0.01, 1000)
E[RMS] is 6.0833 for A, B = (0.1, 1)
E[RMS] is 10.0961 for A, B = (0.1, 10)
E[RMS] is 21.287 for A, B = (0.1, 100)
E[RMS] is 40.5709 for A, B = (0.1, 1000)
E[RMS] is 4.6587 for A, B = (1, 1)
E[RMS] is 4.6516 for A, B = (1, 10)
E[RMS] is 4.814 for A, B = (1, 100)
E[RMS] is 18.1366 for A, B = (1, 1000)
E[RMS] is nan for A, B = (10, 1)
E[RMS] is nan for A, B = (10, 10)
E[RMS] is 4.763 for A, B = (10, 100)
E[RMS] is 4.6625 for A, B = (10, 1000)
E[RMS] is nan for A, B = (100, 1)
E[RMS] is nan for A, B = (100, 10)
E[RMS] is nan for A, B = (100, 100)
E[RMS] is 4.9265 for A, B = (100, 1000)
*** Trivial regressor ***
E[RMS] for the trivial model is 18.867
```

[ ]:

```

```