Assignment IV: Linear Regression Models

Introduction to Machine Learning Lab (190.013), SS2023
Björn Ellensohn[1]

[1] *m01435615, bjoern.ellensohn@stud.unileoben.ac.at, Montanuniversität Leoben, Austria*

May 12, 2023

This document guides through the process of solving Assignment 3.

1 Introduction

The 4th assignment's task was to implement linear regression models on a provided 'diabetes' dataset. the linear regression models had to be implemented from scratch using only numpy and pandas. the models to create where:

- Least Squares Regression
- Ridge Regression
- Lasso Regression

The laste one was a bonus point task. I will walk through the code in 4 Sections.

2 Part I - Abstract Class

2.1 Preparation

Alongside the 'diabetes.csv' dataset there was also an unfinished Jupyter Notebook provided, which gave an outline on how to solve this task. At the begin, there is an abstract class `Predictor`, that should act as a template for the child classes, implementing the actual regression models. However, I decided to extend the metaclass to include the methods that where the same for all child classe. So I only had to write them once. In my oppinion, this is better for the code's structure and enhances readability.

2.2 Defining the Abstract Class

The class `Predictor` is created that provides the basic outline for the main classes of this exercise. The following functions are inlcuded:

- Reading the csv
- Loading and splitting the data into training and test sets
- Normalizing the data

So the dateset is loaded and will be preprocessed accordingly on initiation of the class. The needed data and paramters are stored as atributes.

To mention here are the data loading and pre processing methods. For loading the file, pandas does a good already. The pre-processing was trickier to implement. Datasets tend to have several flaws. Amongst them are outliers and missing values. Missing data can be found with the pandas DataFrame functions `isnan` and `isnull`. The outliers of the dataset where caught by boxplotting every column. In this case, the "1.5 x IQR rule" came to shine, as this allows to automate the process of finding the outliers for each of the columns. The last method is the `train_test_split()` method. Here I want to split the dataset into features and targets, as well as dividing the whole dataset into a test set and a train set. This is especially useful for having a refernce for evaluating the performance later.

3 Part II - Least Squares Regression

In the second part of this exercise, a class `LinearRegression(Predictor)` was created to provide the actual implmentation of the linear regression.

A formula for Least Squares Regression looks like this: `Y = b0 + b1*X1 + b2*X2 + ... + bn*Xn`, where Y is the response variable, b0 is the y-intercept, bi is the coefficient for predictor variable Xi, and Xi is the ith predictor variable.

For evaluation, the predicted data was compared to the test set, so the Mean Squared Error was measured, as well as the R-Squared.

The results are as follows. For comparison, also the unprocessed dataset was evaluated. We can already see, how bad the unprocessed set performs.

```
Least Squares Regression

rg = LinearRegression('diabetes.csv')
# Raw Dataset
rg.regression_raw()
✓ 0.1s

Mean squared error: 3460.083533525722
R-squared: 0.37854964415469516

# Pre-processed Dataset
rg.regression_pp()
✓ 0.0s

Mean squared error: 0.5359087499312691
R-squared: 0.42794122908248045
```

Figure 1: *Results of Least Squares Regression*

4 Part III - Ridge Regression

Next, a class `RidgeRegression(Predictor)` was made to implement Ridge Regression. Ridge Regression is a regularized linear regression model. The formula looks like this:

Again, the model was evaluated against the test set. The results are as follows:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}|\theta_j|$$

Figure 2: *Formula of Ridge Regression*

```
Ridge Regression

rr = RidgeRegression('diabetes.csv')
#Raw Dataset
rr.regression_raw()
✓ 0.1s

Predicted target value: 208.82139795491315
Mean squared error (MSE): 3369.602933909638
R-squared: 0.39480046592928286
Root mean squared error (RMSE): 58.04828105904289

# Pre-processed Dataset
rr.regression_pp()
✓ 0.0s

Predicted target value: 0.7839100243805647
Mean squared error (MSE): 0.5407373522105846
R-squared: 0.42278691076707886
Root mean squared error (RMSE): 0.7353484563188969
```

Figure 3: *Results of Ridge Regression*

5 Part IV - Lasso Regression (Bonus)

The Lasso Regression is also a regularized linear regression model. The formula is like this:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Figure 4: *Formula of Lasso Regression*

Also, the resulting test perfomance was evaluated.

Lasso Regression

```python
lr = LassoRegression('diabetes.csv')
# Raw Dataset
lr.regression_raw()
```

✓ 0.3s

```
Lasso regression:
Mean squared error (MSE): 3574.2278339827644
R-squared: 0.5172053348571604
Root mean sqaured error (RMSE): 59.784846190174015
```

```python
# Pre-processed Dataset
lr.regression_pp()
```

✓ 0.0s

```
Lasso regression:
Mean squared error (MSE): 0.5080018069655786
R-squared: 0.5404721520708679
Root mean sqaured error (RMSE): 0.7127424548640123
```

Figure 5: *Results of Lasso Regression*

APPENDIX

This section houses the code.

Code for Part I

```python
class Predictor:
    def __init__(self, dataset):
        self.coefficients = None
        self.df = pd.read_csv(dataset)
        self.df_pp = self.preprocess(self.df)
        self.X_train_raw, self.X_test_raw, self.y_train_raw, self.y_test_raw = self.train_test_split(self.df)
        self.X_train_pp, self.X_test_pp, self.y_train_pp, self.y_test_pp = self.train_test_split(self.df_pp)

    def preprocess(self, df):
        # It is better to do this after splitting the dataset. --> need to change that
        # Handle missing values
        df.replace(0, np.nan, inplace=True)

        # Remove outliers using iqr rule:
        df_cleaned = df.copy()
        for column in df:
            q1 = df[column].quantile(q=0.25)
            q3 = df[column].quantile(q=0.75)
            med = df[column].median()

            iqr = q3 - q1
            upper_bound = q3+(1.5*iqr)
            lower_bound = q1-(1.5*iqr)

            df_cleaned[column][(df[column] <= lower_bound) | (df[column] >= upper_bound)] = df_cleaned[column].median()

        # Normalize data:
        df_normalized = df_cleaned.copy()
        for column in df_cleaned:

            mean = df_cleaned[column].mean()
            std = df_cleaned[column].std()

            df_normalized[column] = (df_cleaned[column] - mean) / std
        df_processed = df_normalized
        return df_processed

    def train_test_split(self, df, test_size=0.2):
        # Shuffle the rows of the dataset randomly
        df_randomized = df.sample(frac=1, random_state=42).reset_index(drop=True)

        # Extract the features and target variable
        X = df_randomized.drop('target', axis=1)
        y = df_randomized['target']

        # Split the dataset into training and testing sets
        split_ratio = 1 - test_size
        split_index = int(split_ratio * len(df_randomized))

        X_train = X[:split_index]
        X_test = X[split_index:]
        y_train = y[:split_index]
        y_test = y[split_index:]

```

```python
        return X_train, X_test, y_train, y_test

    def fit(self, X, y):
        pass

    def predict(self, X):
        pass
```

Code for Part II

```python
class LinearRegression(Predictor):
    def __init__(self, dataset):
        super().__init__(dataset)

    def fit(self, X, y):
        X = np.insert(X, 0, 1, axis=1)
        y = y.values.reshape(-1, 1)
        self.coefficients = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)

    def predict(self, X):
        X = np.insert(X, 0, 1, axis=1)
        return X.dot(self.coefficients)

    def regression_raw(self):
        # copy the values
        X_train = self.X_train_raw
        y_train = self.y_train_raw
        X_test = self.X_test_raw
        y_test = self.y_test_raw

        # implement linear regression

        # Implement the formula for the least-squares regression line
        X_train_T = np.transpose(X_train)
        beta = np.linalg.inv(X_train_T.dot(X_train)).dot(X_train_T).dot(y_train) # these are the weights

        # Train the model on the training set using the least-squares regression line
        y_pred_train = X_train.dot(beta) # the prediction on the train set

        # Evaluate the performance of the model on the testing set using metrics such as mean squared error and R-squared
        y_pred_test = X_test.dot(beta) # the prediction on the test set

        # calculate the mean squared error
        mse = np.mean((y_test - y_pred_test)**2)
        r_squared = 1 - (np.sum((y_test - y_pred_test)**2) / np.sum((y_test - np.mean(y_test))**2))

        print('Mean squared error:', mse)
        print('R-squared:', r_squared)

        # that should do

    def regression_pp(self):
        # copy the values
        X_train = self.X_train_pp
        y_train = self.y_train_pp
        X_test = self.X_test_pp
        y_test = self.y_test_pp

        # implement linear regression

```

```python
        # Implement the formula for the least-squares regression line
        X_train_T = np.transpose(X_train)
        beta = np.linalg.inv(X_train_T.dot(X_train)).dot(X_train_T).dot(y_train) # these are the weights

        # Train the model on the training set using the least-squares regression line
        y_pred_train = X_train.dot(beta) # the prediction on the train set

        # Evaluate the performance of the model on the testing set using metrics such as mean squared error and R-squared
        y_pred_test = X_test.dot(beta) # the prediction on the test set

        # calculate the mean squared error
        mse = np.mean((y_test - y_pred_test)**2)
        r_squared = 1 - (np.sum((y_test - y_pred_test)**2) / np.sum((y_test - np.mean(y_test))**2))

        print('Mean squared error:', mse)
        print('R-squared:', r_squared)
```

Code for Part III

```python
class RidgeRegression(Predictor):
    def __init__(self, dataset, alpha=1):
        super().__init__(dataset) # this command initializes the parent class (Predictor) and passes the dataset.
        self.alpha = alpha

    def regression_raw(self):
        """
        Fits a ridge regression model on the training data using the specified regularization parameter alpha.
        Using raw dataset
        """
        X_train = self.X_train_raw
        X_test = self.X_test_raw
        y_train = self.y_train_raw
        y_test = self.y_test_raw
        alpha = self.alpha

        # Add a cloumn of 1s to the training data to have the correct dimension.
        X_train = np.hstack([np.ones((X_train.shape[0], 1)), X_train])

        n_features = X_train.shape[1]
        I = np.eye(n_features)
        w = np.linalg.inv(X_train.T.dot(X_train) + alpha * I).dot(X_train.T).dot(y_train)

        #X_test = (X_test - X.mean()) / X.std() #this should not be needed, since the normalization is happening in the constructor
        X_test = np.hstack([np.ones((X_test.shape[0], 1)), X_test])
        y_pred = X_test.dot(w)

        # calculate the mean squared error
        mse = np.mean((y_test - y_pred)**2)
        r_squared = 1 - (np.sum((y_test - y_pred)**2) / np.sum((y_test - np.mean(y_test))**2))

        # calculate root mean squared error (RMSE)
        rmse = np.sqrt(mse)

        print("Predicted target value:", y_pred[0])
        print("Mean squared error (MSE):", mse)
        print("R-squared:", r_squared)
        print("Root mean squared error (RMSE):", rmse)
```

```python

    def regression_pp(self):
        """
        Fits a ridge regression model on the training data using the specified regularization
    ↪ parameter alpha.
        Using processed dataset.
        """
        X_train = self.X_train_pp
        X_test = self.X_test_pp
        y_train = self.y_train_pp
        y_test = self.y_test_pp
        alpha = self.alpha

        # Add a cloumn of 1s to the training data to have the correct dimension.
        X_train = np.hstack([np.ones((X_train.shape[0], 1)), X_train])

        n_features = X_train.shape[1]
        I = np.eye(n_features)
        w = np.linalg.inv(X_train.T.dot(X_train) + alpha * I).dot(X_train.T).dot(y_train)

        X_test = np.hstack([np.ones((X_test.shape[0], 1)), X_test])
        y_pred = X_test.dot(w)

        # calculate the mean squared error
        mse = np.mean((y_test - y_pred)**2)
        r_squared = 1 - (np.sum((y_test - y_pred)**2) / np.sum((y_test - np.mean(y_test))**2))

        # calculate root mean squared error (RMSE)
        rmse = np.sqrt(mse)

        print("Predicted target value:", y_pred[0])
        print("Mean squared error (MSE):", mse)
        print("R-squared:", r_squared)
        print("Root mean squared error (RMSE):", rmse)
```

Code for Part IV

```python
class LassoRegression(Predictor):
    def __init__(self, dataset, alpha=1, max_iter=1000, tol=0.0001):
        super().__init__(dataset)
        self.alpha = alpha
        self.max_iter = max_iter
        self.tol = tol

    def regression_raw(self):
        """
        Fits a lasso regression model on the training data using the specified regularization
    ↪ parameter alpha, iterations, and tolerance.
        Using raw dataset.
        """
        # Define hyperparameters
        alpha = self.alpha  # regularization strength
        max_iterations = self.max_iter  # number of gradient descent iterations
        tolerance = self.tol

        # Load the data
        diabetes = pd.read_csv("diabetes.csv")

        diabetes.insert(0, "Intercept", 1)

        train_size = int(0.8 * len(diabetes))

```

```python
        X_train = diabetes.iloc[:train_size, :-1].values
        y_train = diabetes.iloc[:train_size, -1].values
        X_test = diabetes.iloc[train_size:, :-1].values
        y_test = diabetes.iloc[train_size:, -1].values

        theta_lasso = np.zeros(X_train.shape[1])
        for i in range(max_iterations):
            theta_prev = theta_lasso.copy()
            for j in range(X_train.shape[1]):
                if j == 0:
                    theta_lasso[j] = np.mean(y_train)
                else:
                    xj = X_train[:, j]
                    rj = y_train - X_train @ theta_lasso + xj * theta_lasso[j]
                    zj = xj @ xj
                    if zj == 0:
                        theta_lasso[j] = 0
                    else:
                        if np.sum(xj * rj) > alpha / 2:
                            theta_lasso[j] = (np.sum(xj * rj) - alpha / 2) / zj
                        elif np.sum(xj * rj) < - alpha / 2:
                            theta_lasso[j] = (np.sum(xj * rj) + alpha / 2) / zj
                        else:
                            theta_lasso[j] = 0
            if np.sum((theta_lasso - theta_prev) ** 2) < tolerance:
                break

        sst = np.sum((y_test - np.mean(y_test)) ** 2)

        y_pred_lasso = X_test @ theta_lasso
        mse_lasso = np.mean((y_test - y_pred_lasso) ** 2)
        ssr_lasso = np.sum((y_pred_lasso - np.mean(y_test)) ** 2)
        r_squared_lasso = 1 - (ssr_lasso / sst)

        rmse_lasso = np.sqrt(mse_lasso)

        print("Lasso regression:")
        print("Mean squared error (MSE):", mse_lasso)
        print("R-squared:", r_squared_lasso)
        print("Root mean sqaured error (RMSE):", rmse_lasso)

    def regression_pp(self):
        """
        Fits a lasso regression model on the training data using the specified regularization parameter alpha, iterations, and tolerance.
        Using processed dataset.
        """
        # Define hyperparameters
        alpha = self.alpha  # regularization strength
        max_iterations = self.max_iter  # number of gradient descent iterations
        tolerance = self.tol

        # Load the data
        diabetes_norm = pd.read_csv("diabetes_norm.csv")

        diabetes_norm.insert(0, "Intercept", 1)

        train_size = int(0.8 * len(diabetes_norm))

        X_train = diabetes_norm.iloc[:train_size, :-1].values
        y_train = diabetes_norm.iloc[:train_size, -1].values
        X_test = diabetes_norm.iloc[train_size:, :-1].values
        y_test = diabetes_norm.iloc[train_size:, -1].values

```

```
        theta_lasso = np.zeros(X_train.shape[1])
        for i in range(max_iterations):
            theta_prev = theta_lasso.copy()
            for j in range(X_train.shape[1]):
                if j == 0:
                    theta_lasso[j] = np.mean(y_train)
                else:
                    xj = X_train[:, j]
                    rj = y_train - X_train @ theta_lasso + xj * theta_lasso[j]
                    zj = xj @ xj
                    if zj == 0:
                        theta_lasso[j] = 0
                    else:
                        if np.sum(xj * rj) > alpha / 2:
                            theta_lasso[j] = (np.sum(xj * rj) - alpha / 2) / zj
                        elif np.sum(xj * rj) < - alpha / 2:
                            theta_lasso[j] = (np.sum(xj * rj) + alpha / 2) / zj
                        else:
                            theta_lasso[j] = 0
            if np.sum((theta_lasso - theta_prev) ** 2) < tolerance:
                break

        sst = np.sum((y_test - np.mean(y_test)) ** 2)

        y_pred_lasso = X_test @ theta_lasso
        mse_lasso = np.mean((y_test - y_pred_lasso) ** 2)
        ssr_lasso = np.sum((y_pred_lasso - np.mean(y_test)) ** 2)
        r_squared_lasso = 1 - (ssr_lasso / sst)

        rmse_lasso = np.sqrt(mse_lasso)

        print("Lasso regression:")
        print("Mean squared error (MSE):", mse_lasso)
        print("R-squared:", r_squared_lasso)
        print("Root mean sqaured error (RMSE):", rmse_lasso)
```