Regression with Linear Algebra - Lab

Introduction

In this lab, you'll apply regression analysis using simple matrix manipulations to fit a model to given data, and then predict new values for previously unseen data. You'll follow the approach highlighted in the previous lesson where you used NumPy to build the appropriate matrices and vectors and solve for the β (unknown variables) vector. The beta vector will be used with test data to make new predictions. You'll also evaluate the model fit. In order to make this experiment interesting, you'll use NumPy at every single stage of this experiment, i.e., loading data, creating matrices, performing train-test split, model fitting, and evaluation.

Objectives

In this lab you will:

- Use matrix algebra to calculate the parameter values of a linear regression

First, let's import necessary libraries:

```
import csv # for reading csv file
import numpy as np
```

Dataset

The dataset you'll use for this experiment is "**Sales Prices in the City of Windsor, Canada**", something very similar to the Boston Housing dataset. This dataset contains a number of input (independent) variables, including area, number of bedrooms/bathrooms, facilities(AC/garage), etc. and an output (dependent) variable, **price**. You'll formulate a linear algebra problem to find linear mappings from input features using the equation provided in the previous lesson.

This will allow you to find a relationship between house features and house price for the given data, allowing you to find unknown prices for houses, given the input features.

A description of the dataset and included features is available here.

In your repository, the dataset is available as `windsor_housing.csv` . There are 11 input features (first 11 columns):

```
lotsize bedrooms  bathrms  stories      driveway  recroom       fullbase  gashw
airco   garagepl  prefarea
```

and 1 output feature i.e. **price** (12th column).

The focus of this lab is not really answering a preset analytical question, but to learn how you can perform a regression experiment, using mathematical manipulations - similar to the one you performed using `statsmodels` . So you won't be using any `pandas` or `statsmodels` goodness here. The key objectives here are to:

- Understand regression with matrix algebra and
- Mastery in NumPy scientific computation

Stage 1: Prepare data for modeling

Let's give you a head start by importing the dataset. You'll perform the following steps to get the data ready for analysis:

- Initialize an empty list `data` for loading data
- Read the csv file containing complete (raw) `windsor_housing.csv`. Use `csv.reader()` for loading data.. Store this in `data` one row at a time
- Drop the first row of csv file as it contains the names of variables (header) which won't be used during analysis (keeping this will cause errors as it contains text values)
- Append a column of all 1s to the data (bias) as the first column
- Convert `data` to a NumPy array and inspect first few rows

> NOTE: `read.csv()` reads the csv as a text file, so you should convert the contents to float.

```
# Create Empty lists for storing X and y values
data = []

# Read the data from the csv file
with open('windsor_housing.csv') as f:
    raw = csv.reader(f)
    # Drop the very first line as it contains names for columns - not actual data
    next(raw)
    # Read one row at a time. Append one to each row
    for row in raw:
        ones = [1.0]
        for r in row:
            ones.append(float(r))
        # Append the row to data
        data.append(ones)
data = np.array(data)
data[:5,:]
```

```
array([[1.00e+00, 5.85e+03, 3.00e+00, 1.00e+00, 2.00e+00, 1.00e+00,
        0.00e+00, 1.00e+00, 0.00e+00, 0.00e+00, 1.00e+00, 0.00e+00,
        4.20e+04],
       [1.00e+00, 4.00e+03, 2.00e+00, 1.00e+00, 1.00e+00, 1.00e+00,
        0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00,
        3.85e+04],
       [1.00e+00, 3.06e+03, 3.00e+00, 1.00e+00, 1.00e+00, 1.00e+00,
        0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00,
        4.95e+04],
       [1.00e+00, 6.65e+03, 3.00e+00, 1.00e+00, 2.00e+00, 1.00e+00,
        1.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00,
        6.05e+04],
       [1.00e+00, 6.36e+03, 2.00e+00, 1.00e+00, 1.00e+00, 1.00e+00,
        0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00,
        6.10e+04]])
```

Step 2: Perform a 80/20 train-test split

Explore NumPy's official documentation to manually split a dataset using a random sampling method of your choice. Some useful methods are located in the numpy.random library.

- Perform a **random** 80/20 split on data using a method of your choice in NumPy
- Split the data to create `x_train` , `y_train` , `x_test` , and `y_test` arrays
- Inspect the contents to see if the split performed as expected

> Note: When randomly splitting data, it's always recommended to set a seed in order to ensure reproducibility

```
# Set a seed
np.random.seed(42)
# Perform an 80/20 split
# Make array of indices
all_idx = np.arange(data.shape[0])
# Randomly choose 80% subset of indices without replacement for training
training_idx = np.random.choice(all_idx, size=round(546*.8), replace=False)
# Choose remaining 20% of indices for testing
test_idx = all_idx[~np.isin(all_idx, training_idx)]
# Subset data
training, test = data[training_idx,:], data[test_idx,:]

# Check the shape of datasets
print ('Raw data Shape: ', data.shape)
print ('Train/Test Split:', training.shape, test.shape)

# Create x and y for test and training sets
x_train = training[:,:-1]
y_train = training [:,-1]

x_test = test[:,:-1]
y_test = test[:,-1]

# Check the shape of datasets
print ('x_train, y_train, x_test, y_test:', x_train.shape, y_train.shape, x_test.sha
Raw data Shape:  (546, 13)
Train/Test Split: (437, 13) (109, 13)
x_train, y_train, x_test, y_test: (437, 12) (437,) (109, 12) (109,)
```

Step 3: Calculate the `beta`

With X and y in place, you can now compute your beta values with x_train and y_train as:

$\beta = (x_\text{train}^T . x_\text{train})^{-1} . x_\text{train}^T . y_\text{train}$

- Using NumPy operations (transpose, inverse) that we saw earlier, compute the above equation in steps
- Print your beta values

```
# Calculate Xt.X and Xt.y for beta = (XT . X)-1 . XT . y - as seen in previous lessc
Xt = np.transpose(x_train)
XtX = np.dot(Xt,x_train)
Xty = np.dot(Xt,y_train)

# Calculate inverse of Xt.X
XtX_inv = np.linalg.inv(XtX)

# Take the dot product of XtX_inv with Xty to compute beta
beta = XtX_inv.dot(Xty)

# Print the values of computed beta
print(beta)
```

```
[-5.46637290e+03  3.62457767e+00  2.75100964e+03  1.47223649e+04
  5.97774591e+03  5.71916945e+03  5.73109882e+03  3.83586258e+03
  8.12674607e+03  1.33296437e+04  3.74995169e+03  1.01514699e+04]
```

Step 4: Make predictions

Great, you now have a set of coefficients that describe the linear mappings between X and y. You can now use the calculated beta values with the test datasets that we left out to calculate y predictions. Next, use all features in turn and multiply it with this beta. The result will give a prediction for each row which you can append to a new array of predictions.

$$\hat{y} = x\beta = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_m x_m$$

- Create a new empty list (`y_pred`) for saving predictions
- For each row of `x_test` , take the dot product of the row with beta to calculate the prediction for that row
- Append the predictions to `y_pred`
- Print the new set of predictions

```
# Calculate and print predictions for each row of X_test
y_pred = []
for row in x_test:
    pred = row.dot(beta)
    y_pred.append(pred)
```

Step 5: Evaluate model

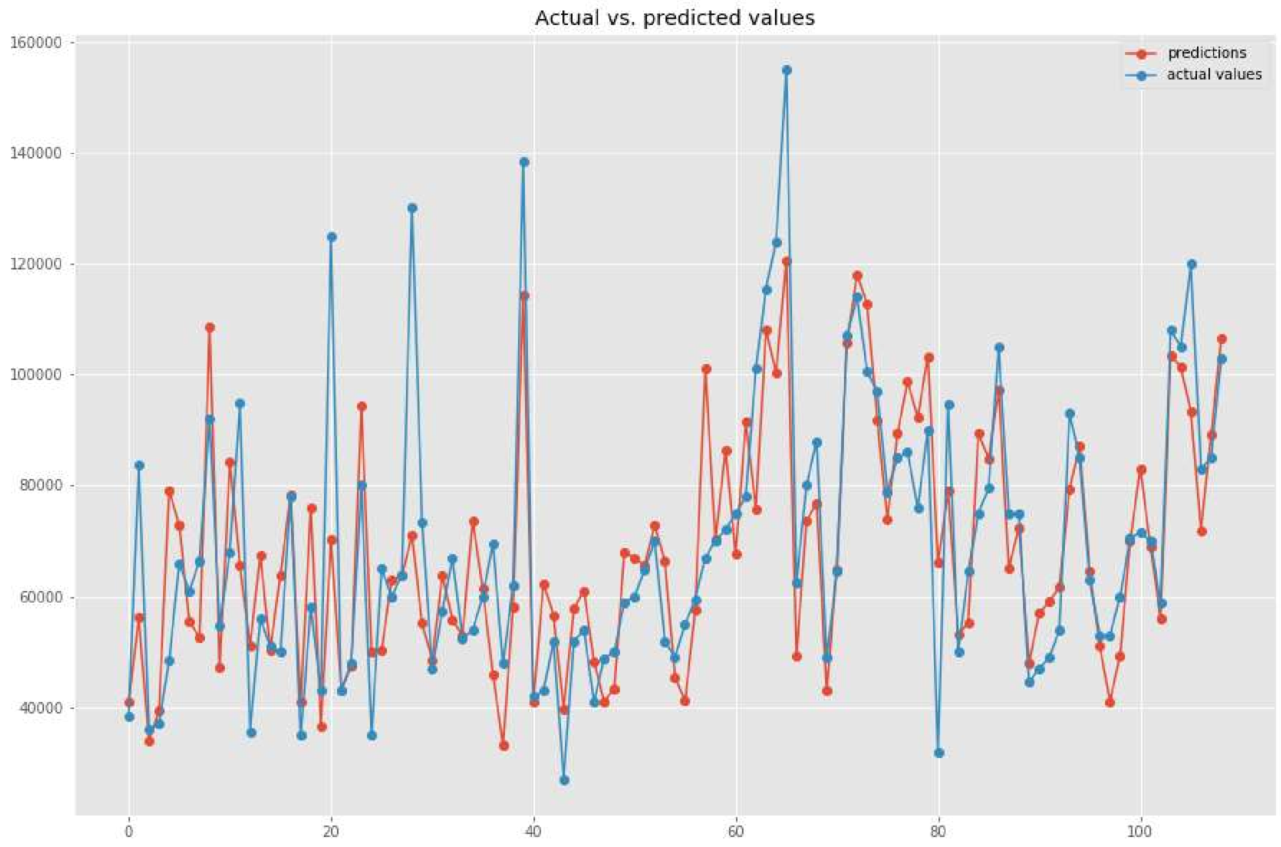
Visualize actual vs. predicted values

This is exciting, now your model can use the beta value to predict the price of houses given the input features. Let's plot these predictions against the actual values in `y_test` to see how much our model deviates.

```
# Plot predicted and actual values as line plots
import matplotlib.pyplot as plt
%matplotlib inline
from pylab import rcParams
rcParams['figure.figsize'] = 15, 10
plt.style.use('ggplot')

plt.plot(y_pred, linestyle='-', marker='o', label='predictions')
plt.plot(y_test, linestyle='-', marker='o', label='actual values')
plt.title('Actual vs. predicted values')
```

```
    plt.legend()
    plt.show()
```



This doesn't look so bad, does it? Your model, although isn't perfect at this stage, is making a good attempt to predict house prices although a few prediction seem a bit out. There could be a number of reasons for this. Let's try to dig a bit deeper to check model's predictive abilities by comparing these prediction with actual values of `y_test` individually. That will help you calculate the RMSE value (root mean squared error) for your model.

Root Mean Squared Error

Here is the formula for RMSE:

$$RMSE = \sqrt{\sum_{i=1}^{N} \frac{(\text{Predicted}_i - \text{Actual}_i)^2}{N}}$$

- Initialize an empty array `err`
- For each row in `y_test` and `y_pred`, take the squared difference and append error for each row in the `err` array
- Calculate $RMSE$ from `err` using the formula shown above

```
# Due to random split, your answers may vary
# Calculate RMSE
err = []
for pred,actual in zip(y_pred,y_test):
    sq_err = (pred - actual) ** 2
    err.append(sq_err)
mean_sq_err = np.array(err).mean()
root_mean_sq_err = np.sqrt(mean_sq_err)
root_mean_sq_err

# Due to random split, your answers may vary
# RMSE = 14868.172645765708
```

```
14868.172645765708
```

Normalized root mean squared error

The above error is clearly in terms of the dependent variable, i.e., the final house price. You can also use a normalized mean squared error in case of multiple regression which can be calculated from RMSE using following the formula:

$$NRMSE = \frac{RMSE}{max_i y_i - min_i y_i}$$

- Calculate normalized RMSE

```
root_mean_sq_err/(y_train.max() - y_train.min())

# Due to random split, your answers may vary
# 0.09011013724706489
```

```
0.09011013724706489
```

There it is. A complete multiple regression analysis using nothing but NumPy. Having good programming skills in NumPy allows you to dig deeper into analytical algorithms in machine learning and deep learning. Using matrix multiplication techniques you saw here, you can easily build a whole neural network from scratch.

Level up (Optional)

- Calculate the R-squared and adjusted R-squared for the above model
- Plot the residuals (similar to `statsmodels`) and comment on the variance and heteroscedasticity
- Run the experiment in `statsmodels` and compare the performance of both approaches in terms of computational cost

Summary

In this lab, you built a predictive model for predicting house prices. Remember this is a very naive implementation of regression modeling. The purpose here was to get an introduction to the applications of linear algebra into machine learning and predictive analysis. There are a number of shortcomings in this modeling approach and you can further apply a number of data modeling techniques to improve this model.

- ● **Jupyter Notebook** 100.0%