Multivariable Linear Regression Comparison

Anshul Choudhary

19-05-25

Abstract

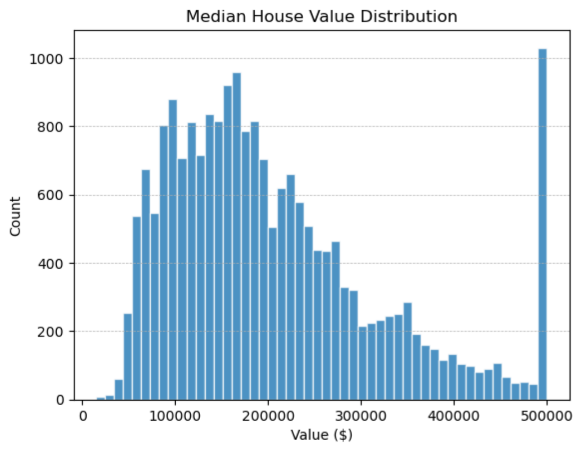
In this assignment, we use the California Housing Dataset to implement three algorithms for predicting house prices.

1 Data Pre-Processing

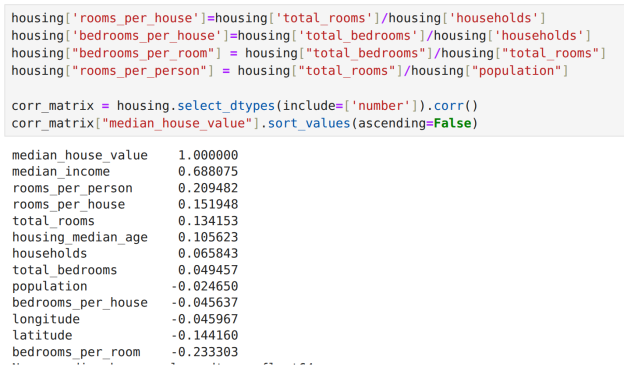
First, let's check what we have been provided in the data.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(9), object(1)
```

- **Cleaning the data**: There are nine features taking float values except the ocean_proximity feature, which is categorical (e.g., INLAND, ISLAND). Since ML models only work with numbers, we use one-hot encoding for this feature via pandas' `get_dummies`. We also notice missing values in `total_bedrooms`. Dropping the entire column would lose information, so we fill missing values with the median.



We notice from the figure that the value of the houses has been capped at 500000, since there is a spike there. This will penalize our models from predicting the median price higher than 500000(even if in reality the model is correct) so we will drop all these capped data points.



```
housing['rooms_per_house']=housing['total_rooms']/housing['households']
housing['bedrooms_per_house']=housing['total_bedrooms']/housing['households']
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["rooms_per_person"] = housing["total_rooms"]/housing["population"]

corr_matrix = housing.select_dtypes(include=['number']).corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

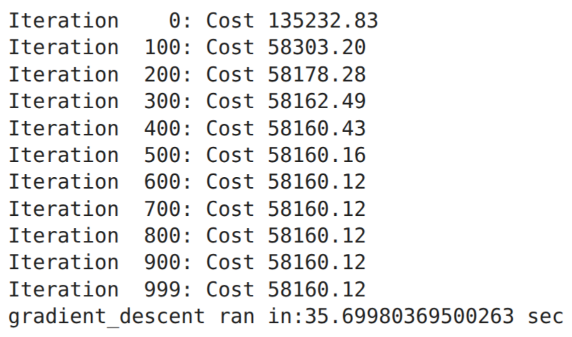
```
median_house_value    1.000000
median_income         0.688075
rooms_per_person      0.209482
rooms_per_house       0.151948
total_rooms           0.134153
housing_median_age    0.105623
households            0.065843
total_bedrooms        0.049457
population           -0.024650
bedrooms_per_house   -0.045637
longitude            -0.045967
latitude             -0.144160
bedrooms_per_room    -0.233303
```

- **Feature engineering**: The data represents blocks of houses, so parameters like `total_rooms` refer to an entire block. I created features such as rooms per household and rooms per person. From the correlation matrix, these engineered features correlate better with house value than the original ones. Also, `households` correlates 0.97 with `total_bedrooms`, so I dropped `total_bedrooms` (households has a stronger correlation with price).
- **Splitting the data**: I split the data into training (80%) and validation (20%) sets.

Finally, I scaled the features using Z-score normalization to help gradient descent converge faster. This same processed data is used in all three approaches to ensure a fair comparison.

2 Results

2.1 Pure-Python Implementation

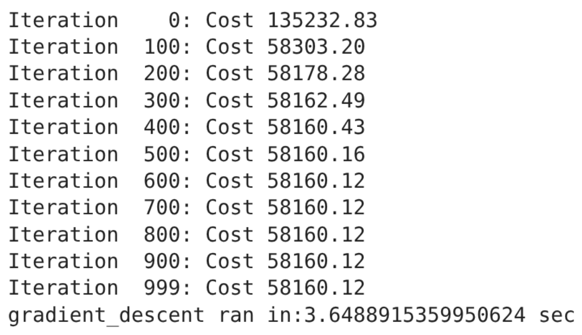


```
Iteration    0: Cost 135232.83
Iteration  100: Cost 58303.20
Iteration  200: Cost 58178.28
Iteration  300: Cost 58162.49
Iteration  400: Cost 58160.43
Iteration  500: Cost 58160.16
Iteration  600: Cost 58160.12
Iteration  700: Cost 58160.12
Iteration  800: Cost 58160.12
Iteration  900: Cost 58160.12
Iteration  999: Cost 58160.12
gradient_descent ran in:35.69980369500263 sec
```

Figure 1: Convergence of RMSE

We can observe that the model converges in about 500 iterations and takes about 35.69 seconds.

2.2 NumPy Implementation

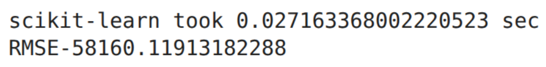


```
Iteration    0: Cost 135232.83
Iteration  100: Cost 58303.20
Iteration  200: Cost 58178.28
Iteration  300: Cost 58162.49
Iteration  400: Cost 58160.43
Iteration  500: Cost 58160.16
Iteration  600: Cost 58160.12
Iteration  700: Cost 58160.12
Iteration  800: Cost 58160.12
Iteration  900: Cost 58160.12
Iteration  999: Cost 58160.12
gradient_descent ran in:3.6488915359950624 sec
```

Figure 2: Convergence of RMSE

This model also converges in about 500 iterations and takes about 3.64 seconds—roughly 10× faster than the pure-Python code.

2.3 Scikit-Learn Implementation



```
scikit-learn took 0.027163368002220523 sec
RMSE-58160.11913182288
```

Figure 3: RMSE with scikit-learn

The fitting duration is 0.023 seconds.

3 Convergence Plots

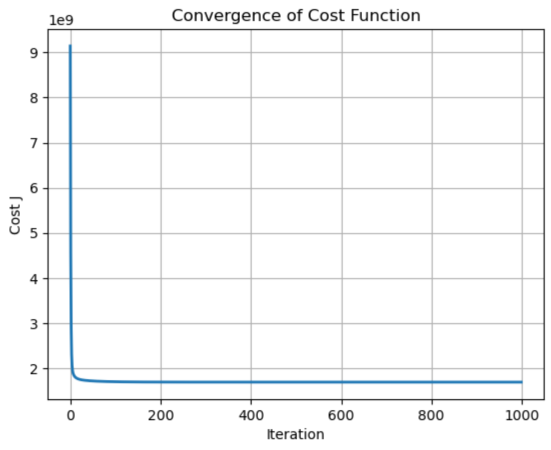


Figure 4: Convergence of Pure-Python approach

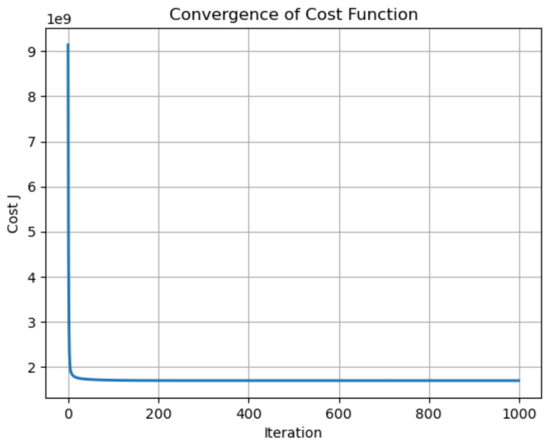


Figure 5: Convergence of NumPy approach

Both plots are identical, since data and algorithm are the same; only the execution time differs.

3.1 Runtime Comparison

Method	Time (s)
Pure Python	35.69
NumPy	3.648
scikit-learn	0.027

Table 1: Comparison of run-times

3.2 Final Metrics

Method	RMSE	MAE	R^2
Pure Python	58160.119	42404.575	0.642
NumPy	58160.119	42404.575	0.642
scikit-learn	58160.119	42404.590	0.642

Table 2: Performance metrics (rounded to three decimals) on the **training set**.

Method	RMSE	MAE	R^2
Pure Python	62644.640	45210.297	0.601
NumPy	62644.640	45210.297	0.601
scikit-learn	62644.640	45210.321	0.601

Table 3: Performance metrics (rounded to three decimals) on the **validation set**.

The performance on training set and validation set is very similar, this implies the models are not overfitting, though the huge error might mean that the models are underfitting.

4 Analysis

- **Performance metrics**: As we can see, the final performance metrics for all three methods are identical. This is expected because they use the same preprocessed data. The pure-Python and NumPy implementations are initialized with the same weights and use the same optimization algorithm (gradient descent); vectorization affects only runtime, not the final weight vector. Scikit-Learn uses ordinary least squares (OLS), computing the solution directly without iterations, so it is independent of the initial weights. Since mean squared error has a single global minimum, all methods converge to the same point.

- **Speed-up factors**: The pure-Python and NumPy models differ only in convergence time (35.69s vs 3.64s). The reason for the enhanced performance of the NumPy model is because vectorization enables batch processing of data using optimized, low-level implementations that reduce Python-level loops and overhead. Ad- ditionally, many NumPy operations use parallelization allowing computations to run concur- rently on multiple CPU cores. Together, vectorization and parallelization significantly speed up numerical computations, leading to faster convergence. Scikit-Learn's 0.02s is due to its highly optimized OLS solver.

- **Scalability trade-offs**: Although Scikit-Learn's OLS solver was very fast on our dataset, it may struggle with larger datasets as the number of features grows. OLS requires fitting the entire dataset into memory; when memory is limited, SGD(stochastic gradient descent) is preferable. The time complexity of the pure-Python and NumPy implementations is $O(IND)$ (iterations I, samples N, features D), whereas Scikit-Learn's OLS has $O(ND^2)$ complexity.

- **Initial parameters and learning rate**: Different weight initializations in the pure-Python and NumPy methods will lead to the same final model, since both converge to the single global minimum of the convex cost surface, though the iteration count might vary depending on which initialization is closer to the minima. Choosing the right learning rate (α) is crucial: I tried various values $\alpha = 1$ diverged, $\alpha = 0.1$ converged slowly (4000–5000 iterations), and $\alpha = 0.4$ converged in about 500-600 iterations.