

# Predicting Housing Prices Using Linear Regression

[https://colab.research.google.com/drive/1un4nIb\\_kJpaUJhy7-  
kHFFxLcmzGc23Bb?usp=sharing](https://colab.research.google.com/drive/1un4nIb_kJpaUJhy7-kHFFxLcmzGc23Bb?usp=sharing)

*John-Paul Martin*

# Table of Contents

Motivation.....	3
Introduction.....	3
Understanding the ML Framework.....	5
Hypothesis.....	5
Loss Function.....	6
Optimisation.....	6
Regularisation .....	7
Evaluation Metrics .....	7
Model Family.....	8
Advantages and Disadvantages.....	8
Training Process.....	8
Implementation .....	9
Simple Linear Regression.....	9
Multiple Linear Regression (Gradient Descent) .....	10
MLR (Stochastic Gradient Descent) .....	11
MLR (Stochastic Gradient Descent) + L1 Regularisation.....	12
MLR (Stochastic Gradient Descent) + L2 Regularisation.....	13
References.....	14
Appendix.....	14

# Motivation

I chose to study the linear regression model as it is one of the most used models in the Data Science industry and provides a strong foundation for understanding relationships between variables. As an aspiring data scientist, it felt natural to start with linear regression for its simplicity and interpretability. Studying this model also gives me practical experience in building, evaluating, and visualising predictions, which are essentials skills for real world data analysis.

I chose the problem of predicting housing prices because of the current challenges in the Australian housing market, where affordability has become a growing concern, particularly for younger Australians trying to enter the property market. With rising prices and limited options, there is an increasing need for models that can provide reliable predictions to help identify more accessible housing options. Linear regression is particularly well-suited to this problem because it excels at predicting numerical outcomes, making it an appropriate choice for this analysis.

# Introduction

To initiate this project, I was required to choose between two options, ultimately selecting Option 1 due to my interest in the linear regression model. Following this, my focus shifted to gaining a thorough understanding of the models input and output formats, a critical step in ensuring its effective application.

As a supervised learning algorithm, linear regression uses an input variable  $x$  (e.g. numeric features such as age, income, or temperature), and an output variable  $y$  (numeric prediction such as sales figures or temperature forecasts), assuming a linear relationship between them. This relationship is represented by a straight line, suggesting changes in the input variable directly influence the output variable, a concept realised by the equation:

- $y = mx + c$ , where  $m$  is the slope and  $c$  is the intercept

To reinforce my learning, I leveraged AI assistance to develop a dummy function, allowing me to experiment with the model's mechanics and visualise how data input translates into predicted outputs.

```

from typing import List

def predict_price(features: List[float]) -> float:
    """
    Dummy linear regression-like function.
    Takes a list of numeric features and returns a predicted numeric value.
    """

    # Completely fake "weights"
    weights = [2.0, -1.0, 0.5] # pretend we have 3 features
    bias = 10.0 # pretend intercept

    # If the user passes more or fewer features than our dummy weights
    # we'll just cut or pad them to fit
    features = features[:len(weights)] + [0] * (len(weights) - len(features))

    # Fake prediction = weighted sum of features + bias
    prediction = sum(w * x for w, x in zip(weights, features)) + bias

    return prediction

```

*Figure 1. AI generated dummy function*

After this, I selected a suitable dataset that aligned well with the requirements of linear regression models. I sourced a dataset online from Kaggle, specifically the [Melbourne Housing Snapshot](#), which became the foundation for my task of predicting housing prices. Once a dataset was sourced, I utilised AI again to help preprocess the data to make it usable on the provided dummy function and produce results.

```

def load_and_preprocess_stock_data(filepath: str):
    """
    Loads stock dataset from CSV and prepares features/target for regression.
    """
    # Load CSV
    df = pd.read_csv('/content/drive/MyDrive/ml_2025/all_stocks_5yr.csv')

    # Ensure sorted by date
    df['Date'] = pd.to_datetime(df['date'])
    df = df.sort_values('date')

    # Select numeric features (example: Open, High, Low, Close, Volume)
    feature_cols = ['open', 'high', 'low', 'close', 'volume']
    X = df[feature_cols].copy()

    # Target: next day's Close price
    y = df['close'].shift(-1) # shift up so today's row predicts tomorrow's close

    # Remove last row (target is NaN there)
    X = X.iloc[:-1].values
    y = y.iloc[:-1].values

    return X, y

```

*Figure 2. Dummy Function Pre-Processing*

```

[1]: if __name__ == "__main__":
    X, y = load_and_preprocess_stock_data("/content/drive/MyDrive/ml_2025/all_stocks_5yr.csv")

    # Test first 5 predictions
    for i in range(5):
        features = X[i].tolist()
        prediction = predict_next_day(features)
        print(f"Features: {features}")
        print(f"Predicted: {prediction:.2f}, Actual: {y[i]:.2f}\n")

[2]: Features: [15.07, 15.12, 14.63, 14.75, 8407500.0]
Predicted: 420482.65, Actual: 81.89

Features: [81.22, 81.93, 80.94, 81.89, 296853.0]
Predicted: 14983.55, Actual: 79.07

Features: [78.24, 79.07, 78.125, 79.07, 4632684.0]
Predicted: 231773.60, Actual: 238.16

Features: [236.64, 238.6924, 235.75, 238.16, 552207.0]
Predicted: 27829.50, Actual: 89.16

Features: [89.04, 89.48, 88.91, 89.16, 554948.0]
Predicted: 27892.06, Actual: 22.68

```

*Figure 3. Dummy Function Results*

This brief demonstration provided me with a valuable opportunity to experiment and thoroughly understand the inner workings of a simple linear regression model, giving me a hands-on appreciation of its functionality and predictive capabilities.

## Understanding the ML Framework

### Hypothesis

The hypothesis represents the equation employed to predict the dependant variables ( $y$ ) based on the independent variable ( $x$ ). In a straightforward scenario involving a single independent variable, this relationship can be expressed as:

- $h(x) = \beta_0 + \beta_1 x \rightarrow$  or more simply,  $y = mx + b$ :
  - $h(x)$  denotes the predicted value
  - $\beta_0$  serves as the bias / intercept
  - $\beta_1$  acts as the weight / slope
  - $x$  is the independent variable

For a more common, complex scenario in multiple linear regression where we have more than one independent variable:

- $h(x_1, x_2, \dots, x_k) = \beta_0 + \beta_1x_1 + \beta_2x_2 + \dots + \beta_kx_k$  where:
  - $h(x_1, x_2, \dots, x_k)$  represents the combination of independent variables
  - $\beta_0$  remains the bias or intercept
  - $\beta_1x_1 + \beta_2x_2 + \dots + \beta_kx_k$  indicates the coefficients or weights / slopes associated with each independent variable

Therefore, the hypothesis space of a linear regression model is the set of all possible linear functions the model can learn to map input features to an output. In other words, every straight line relationship that could be drawn between inputs and the target.

## Loss Function

The loss function quantifies the discrepancy between the models' predictions and the actual values, helping to refine the model's line for an optimal fit. To begin, it is essential to understand how error is computed. The formula for this is:

- $\text{error} = y_i - \hat{y}_i$  where:
  - $y_i$  represents the actual values
  - $\hat{y}_i$  denotes the predicted values

This error value forms the basis for various loss functions, including the widely adopted Mean Squared Error (MSE) in linear regression. MSE stands out due to its method of squaring the errors, which prevents negative and positive values from offsetting each other and gives a stronger penalty for larger errors. This characteristic makes MSE particularly suitable for enhancing the accuracy of linear regression models. The formula for MSE is:

$$\text{MSE} = \frac{1}{n} \sum (Y_i - \hat{Y}_i)^2$$

*Figure 3. Formula for MSE*

## Optimisation

The goal of optimisation in linear regression is to find the best values for  $\beta_0$  and  $\beta_1$  to minimise the MSE and achieve the optimal fitting line. Optimisation techniques involve iteratively adjusting these parameters to reduce prediction errors, navigating the error landscape to find the lowest MSE.

**Gradient Descent**, a key method, uses the loss functions gradient to guide this process towards the best fit. It is an iterative technique that:

1. starts with initial parameter values
2. compute gradient of loss function using all training samples
3. update parameters in the opposite direction of the gradient (opposite since we want to reduce the gradient to get the lowest possible loss value)
4. scale the update by the learning rate
5. repeat until the MSE is minimised and the best fit line is found

Another optimisation technique is **Stochastic Gradient Descent**. It acts the same as regular Gradient Descent, however instead of updating the weights and bias each epoch, SGD instead updates for every sample every epoch. For example, if we had 1000 epochs and n samples, there would be  $1000 * n$  updates.

1. Start with initial parameter values (weights and bias)
2. Loop through each epoch
3. For each sample in training set:
  - a. Compute prediction for single sample
  - b. Calculate the error

- c. Compute gradient of the loss for that single sample
- d. Update parameters in the opposite direction of the gradient
- e. Scale the update by the learning rate

## Regularisation

Although not directly part of the Optimisation phase in the ML Framework, regularisation does optimise models. Regularisation is a technique that adds a penalty to the loss function to discourage overly large weights. This helps the model generalise better to unseen data and avoid overfitting.

### Why is it used in linear regression:

- Standard linear regression tries to minimise the error on the training set
- If there are many features or noisy data, it can assign very large weights, fitting the training data too closely
- Regularisation limits the magnitude of weights, making the model simpler and more robust

### Type of Regularisation:

1. L1 (Lasso): Adds a penalty equal to the sum of the absolute values of the weights
  - Encourages sparsity, some weights can be pushed exactly to zero
  - Useful for feature selection, as it can effectively remove irrelevant features

$$\text{Loss} = \text{MSE} + \lambda \sum_j |\beta_j|$$

2. L2 (Ridge): Adds a penalty equal to the sum of the squares of the weights
  - Shrinks all weights toward zero without making them exactly zero
  - Reduces sensitivity to noisy features

$$\text{Loss} = \text{MSE} + \lambda \sum_j \beta_j^2$$

## Evaluation Metrics

Evaluation metrics are used to assess the performance of a model by quantifying how close predicted values are to actual values. They help determine the model's accuracy, guide improvements, and compare different models or approaches.

Two common metrics are:

1. RMSE: The square root of the MSE, gives higher weights to larger errors, making it sensitive to outliers while remaining in the same units as the target variable.

- MAE (Mean Absolute Error): Calculates the average absolute difference between predicted and actual values, treating all errors equally and providing a clearer picture of typical prediction performance.

## Model Family

The model family for my linear regression model comprises all possible straight-line equations that map a set of house features to predicted house prices. Each model in this family is defined by a specific set of parameters that determine the slope and intercept of the line, allowing it to make predictions for any given house.

- Family:** all linear functions that map input features to predicted outcomes
- Parameters:** the weights (coefficients) for each feature and the bias term, which together define the specific behaviour of a model within the family
- Input format:** a vector of independent variables representing the characteristics of a house, such as size, number of rooms, location, and age
- Output format:** a single numeric value corresponding to the predicted house price

The model is parameterised by weights  $\beta$  and bias  $\beta_0$ , with the parameter space  $R^{n+1}$  where  $n$  is the number of features, and  $R$  is the set of all real numbers. By adjusting these parameters, the model can find the straight-line relationship that best fits the observed data.

## Advantages and Disadvantages

Before applying linear regression to a prediction task, it is important to understand its inherent advantages and disadvantages. Here, I identify its key strengths and weaknesses in relation to my task. Most of the information gathered was sourced from Geeks for Geeks (2025).

### Advantages:

- Simple implementation and interpretable outputs
- Best for linear data due to its simplicity compared to other models
- Can utilise techniques such as dimension reduction, regularisation, and cross validation to avoid overfitting

### Disadvantages:

- Outliers have a heightened effect compared to other models
- Assumes a linear relationship between dependent and independent variables
- The model's reliable worsens if inputs are too similar

## Training Process

To solidify my understanding of the linear regression model, I asked ChatGPT to concisely explain the training process. This produced the following:

1. **Initialise parameters:** set weights for each feature and bias term, often using small random values or zeros
2. **Make predictions:** compute predicted values by taking the weighted sum of input features plus the bias
3. **Calculate errors:** find the difference between the actual and predicted values
4. **Compute loss:** aggregate errors using a loss function to quantify overall prediction error
5. **Optimise parameters:** apply an optimisation method such as gradient descent, which calculate the gradient of the loss with respect to each parameter and updates the weights and bias to reduce error
6. **Iterate:** repeat prediction, error calculation, loss computation, and optimisation across multiple training cycles until loss converges or improvements become minimal
7. **Result:** outputted is a fitted linear equation representing the best fitting line, ready to predict unseen data.

## Implementation

### Simple Linear Regression

1. Prepare data by allocating input feature (Distance) and target variable (Price)
2. Split X and Y into 75% training and 25% testing using an index calculated based on dataset length
3. Initialise parameters:
  - a. Determine number of features
  - b. Beta = weights, start at zeros
  - c. Bias = intercept, start at zero
  - d. Learning\_rate: step size for updates
  - e. Epochs: number of training iterations
4. Training loop (Gradient Descent) – for each epoch:
  - a. Hypothesis:  $\hat{y} = X_{train} \cdot \beta + b$
  - b. Loss function:  $MSE = \text{mean}((y_{train} - \hat{y})^2)$
  - c. Error term:  $e = \hat{y} - y_{train}$
  - d. Compute Gradients:
    - i. For bias:  $db = 2/n \sum e$
    - ii. For weights:  $d\beta = 2/n e \cdot X_{train}$
  - e. Update bias and weights using learning rate and gradient
5. Evaluate on test set:
  - a. RMSE = square root of MSE
  - b. MAE =  $\text{mean}(|\hat{y}_{test} - y_{test}|)$
6. Predict on new samples:
  - a. Create new input (e.g Distance = 5km)
  - b. Apply trained weights and bias to generate predicted price

**Final Results:** RMSE = 485350.94, MAE = 384868.75

## Multiple Linear Regression (Gradient Descent)

1. Prepare data by allocating input features (feature\_cols = 'Rooms', 'Distance', 'Bedroom2', 'Bathroom', 'Car', 'Landsize', 'BuildingArea', 'Latitude', 'Longitude', 'Age', 'Type\_h', 'Type\_t', 'Type\_u') and target variable Price
2. Split X and Y into 75% training and 25% testing using an index calculated based on dataset length
3. Standardise features using training mean and standard deviation, then apply same scaling to test data. This improves model training speed and reliability.
4. Apply log transformation to target variable ( $\log(1+price)$ ) to reduce skewness, stabilise variance, and make data distribution more suitable for regression.
5. Initialise parameters:
  - a. Determine number of features
  - b. Beta = weights, start at zeros
  - c. Bias = intercept, start at zero
  - d. Learning\_rate: step size for updates
  - e. Epochs: number of training iterations
6. Training loop (Gradient Descent) – for each epoch:
  - a. Hypothesis:  $y_{pred\_log} = \text{np.dot}(X_{train\_m}, \beta_m) + \text{bias}_m$
  - b. Error term:  $error_m = y_{pred\_log} - y_{train\_log}$
  - c. Compute Gradients:
    - i. For bias:  $db_m = (2 / \text{len}(y_{train\_log})) * \text{np.sum}(error_m)$
    - ii. For weights:  $d_{\beta_m} = (2 / \text{len}(y_{train\_log})) * \text{np.dot}(X_{train\_m.T}, error_m)$
  - d. Update bias and weights using learning rate and gradient
7. Model evaluation
  - a. Use trained model to predict target values for test data ( $y_{test\_pred\_log} = \text{np.dot}(X_{test\_m}, \beta_m) + \text{bias}_m$ )
  - b. Reverse log transformation by exponentiating so results can be interpreted in target unit
  - c. Calculate average of squared differences between predicted and actual values ( $mse = \text{np.mean}((y_{test\_pred} - y_{test\_m}) ** 2)$ )
  - d. Evaluate using RMSE and MAE
8. Predict on new sample
  - a. Define an array to represent a new house, containing the feature values for a single house we want to predict for
  - b. Scale features (new\_house\_m\_scaled) using the same mean and standard deviation as the training data, ensuring compatibility with the model
  - c. Predict in log scale
  - d. Convert back to dollars by exponentiating
  - e. Display result

**Final Results:** RMSE = 291964.35, MAE = 200398.10

**Adjustments Made:** To introduce variability during training and observe changes in RMSE and MAE, I will experiment with different optimisation approaches and then compare their performance. The next section focuses on Stochastic Gradient Descent.

## MLR (Stochastic Gradient Descent)

1. Prepare data by allocating input features (feature\_cols = 'Rooms', 'Distance', 'Bedroom2', 'Bathroom', 'Car', 'Landsize', 'BuildingArea', 'Latitude', 'Longitude', 'Age', 'Type\_h', 'Type\_t', 'Type\_u') and target variable Price
2. Split X and Y into 75% training and 25% testing using an index calculated based on dataset length
3. Standardise features using training mean and standard deviation, then apply same scaling to test data. This improves model training speed and reliability.
4. Apply log transformation to target variable ( $\log(1+price)$ ) to reduce skewness, stabilise variance, and make data distribution more suitable for regression.
5. Initialise parameters:
  - a. Determine number of features
  - b. Beta = weights, start at zeros
  - c. Bias = intercept, start at zero
  - d. Learning\_rate: step size for updates
  - e. Epochs: number of training iterations
6. Training loop (Stochastic Gradient Descent) – for each epoch, and for each individual sample:
  - a. Select the feature variables for the single sample and its corresponding target variable
  - b. Compute predicted log price for sample employing hypothesis formula ( $y\_pred\_i = np.dot(xi, beta_m) + bias_m$ )
  - c. Calculate error term by subtracting actual values from predicted values
  - d. Compute gradients:
    - i. For bias:  $2 * error\_i$
    - ii. For weights:  $2 * xi * error\_i$
    - iii. Update bias and weights using learning rate and gradients
7. Model evaluation
  - a. Use trained model to predict target values for test data ( $y\_test\_pred\_log = np.dot(X\_test\_m, beta\_m) + bias\_m$ )
  - b. Reverse log transformation by exponentiating so results can be interpreted in target unit
  - c. Calculate average of squared differences between predicted and actual values ( $mse = np.mean((y\_test\_pred - y\_test\_m) ** 2)$ )
  - d. Evaluate using RMSE and MAE
8. Predict on new sample
  - a. Define an array to represent a new house, containing the feature values for a single house we want to predict for

- b. Scale features (`new_house_m_scaled`) using the same mean and standard deviation as the training data, ensuring compatibility with the model
- c. Predict in log scale
- d. Convert back to dollars by exponentiating
- e. Display result

**Final Results:** RMSE = 279663.66, MAE = 196397.30

## MLR (Stochastic Gradient Descent) + L1 Regularisation

1. Prepare data by allocating input features (`feature_cols = 'Rooms', 'Distance', 'Bedroom2', 'Bathroom', 'Car', 'Landsize', 'BuildingArea', 'Latitude', 'Longitude', 'Age', 'Type_h', 'Type_t', 'Type_u'`) and target variable `Price`
2. Split X and Y into 75% training and 25% testing using an index calculated based on dataset length
3. Standardise features using training mean and standard deviation, then apply same scaling to test data. This improves model training speed and reliability.
4. Apply log transformation to target variable ( $\log(1+price)$ ) to reduce skewness, stabilise variance, and make data distribution more suitable for regression.
5. Initialise parameters:
  - a. Determine number of features
  - b. Beta = weights, start at zeros
  - c. Bias = intercept, start at zero
  - d. Learning\_rate: step size for updates
  - e. Epochs: number of training iterations
  - f. Lambda\_11: set regularisation strength to control how strongly the model shrinks weights
6. Training loop (Stochastic Gradient Descent) – for each epoch, and for each individual sample:
  - a. Select the feature variables for the single sample and its corresponding target variable
  - b. Compute predicted log price for sample employing hypothesis formula ( $y_{pred\_i} = np.dot(xi, beta_m) + bias_m$ )
  - c. Calculate error term by subtracting actual values from predicted values
  - d. Compute gradients:
    - i. For bias:  $2 * error_i$
    - ii. For weights:  $2 * xi * error_i + lambda_11 * np.sign(beta_m)$
    - iii. Update bias and weights using learning rate and gradients
7. Model evaluation
  - a. Use trained model to predict target values for test data ( $y_{test\_pred\_log} = np.dot(X_{test\_m}, beta_m) + bias_m$ )
  - b. Reverse log transformation by exponentiating so results can be interpreted in target unit
  - c. Calculate average of squared differences between predicted and actual values ( $mse = np.mean((y_{test\_pred} - y_{test\_m}) ** 2)$ )

- d. Evaluate using RMSE and MAE
- 8. Predict on new sample
  - a. Define an array to represent a new house, containing the feature values for a single house we want to predict for
  - b. Scale features (new\_house\_m\_scaled) using the same mean and standard deviation as the training data, ensuring compatibility with the model
  - c. Predict in log scale
  - d. Convert back to dollars by exponentiating
  - e. Display result

**Final Results:** RMSE = 279912.46, MAE = 196540.49

## MLR (Stochastic Gradient Descent) + L2 Regularisation

1. Prepare data by allocating input features (feature\_cols = 'Rooms', 'Distance', 'Bedroom2', 'Bathroom', 'Car', 'Landsize', 'BuildingArea', 'Latitude', 'Longitude', 'Age', 'Type\_h', 'Type\_t', 'Type\_u') and target variable Price
2. Split X and Y into 75% training and 25% testing using an index calculated based on dataset length
3. Standardise features using training mean and standard deviation, then apply same scaling to test data. This improves model training speed and reliability.
4. Apply log transformation to target variable ( $\log(1+price)$ ) to reduce skewness, stabilise variance, and make data distribution more suitable for regression.
5. Initialise parameters:
  - a. Determine number of features
  - b. Beta = weights, start at zeros
  - c. Bias = intercept, start at zero
  - d. Learning\_rate: step size for updates
  - e. Epochs: number of training iterations
  - f. Lambda\_l2: set regularisation strength to control how strongly the model shrinks weights
6. Training loop (Stochastic Gradient Descent) – for each epoch, and for each individual sample:
  - a. Select the feature variables for the single sample and its corresponding target variable
  - b. Compute predicted log price for sample employing hypothesis formula ( $y_{pred\_i} = \text{np.dot}(x_i, \beta_m) + \beta_0$ )
  - c. Calculate error term by subtracting actual values from predicted values
  - d. Compute gradients:
    - i. For bias:  $2 * \text{error}_i$
    - ii. For weights:  $2 * x_i * \text{error}_i + 2 * \lambda * \beta_m$
    - iii. Update bias and weights using learning rate and gradients
7. Model evaluation
  - a. Use trained model to predict target values for test data ( $y_{test\_pred\_log} = \text{np.dot}(X_{test\_m}, \beta_m) + \beta_0$ )

- b. Reverse log transformation by exponentiating so results can be interpreted in target unit
  - c. Calculate average of squared differences between predicted and actual values  
( $\text{mse} = \text{np.mean}((\text{y\_test\_pred} - \text{y\_test\_m}) ^\star 2)$ )
  - d. Evaluate using RMSE and MAE
8. Predict on new sample
- a. Define an array to represent a new house, containing the feature values for a single house we want to predict for
  - b. Scale features (`new_house_m_scaled`) using the same mean and standard deviation as the training data, ensuring compatibility with the model
  - c. Predict in log scale
  - d. Convert back to dollars by exponentiating
  - e. Display result

**Final Results:** RMSE = 279702.54, MAE = 196430.89

## References

Geeks for Geeks. (2025, July 12). *ML-Advantages and Disadvantages of Linear Regression*.  
<https://www.geeksforgeeks.org/machine-learning/ml-advantages-and-disadvantages-of-linear-regression/>

## Appendix

ChatGPT Conversation: <https://chatgpt.com/share/68e0980c-b660-8007-aa9f-dc6487339aca>

- Gave me a clearer understanding of my chosen model and its machine learning framework
- Guided the step by step development of the models implementation into code

**Example Query:** With AI assistance, try writing a “dummy” function that matches the signature of your selected model—meaning the input/output types match. For example, for a sentiment classification task, you might write: `def classify_sentiment(text_sequence: List[str]) -> str: # find "happy" in text_sequence if "happy" in text_sequence: ret = "positive" else: ret = "negative"` return ret do this but for me i am doing linear regression, predicting sales forecast

**ChatGPT Response:** Got it — since you’re predicting sales forecasts using linear regression, your “dummy” function just needs to take in numeric inputs (features like advertising spend, season index, etc.) and output a numeric prediction (sales).

Here’s a simple Python dummy version:

```
from typing import List

def predict_sales(features: List[float]) -> float:
```

"""

Dummy sales prediction function for linear regression.

Args:

    features (List[float]): A list of numeric features (e.g., marketing spend, season index, etc.)

Returns:

    float: Predicted sales value.

"""

```
# Dummy rule: Just sum the features and add a constant offset
```

```
prediction = sum(features) * 1.5 + 1000 # made-up formula
```

```
return prediction
```

```
# Example usage
```

```
example_features = [2000, 3] # e.g., [marketing spend, season index]
```

```
predicted_sales = predict_sales(example_features)
```

```
print(f'Predicted Sales: {predicted_sales}')
```

This isn't actually learning — it just mimics the same input/output structure that a real trained linear regression model would have.

If you want, I can also make a version that matches your dataset column names. That way, it feels more like your real sales forecasting model.

**Follow up:** now, • Ask AI to build a proper ML model to replace your dummy function.

**Response:** Alright — let's upgrade your dummy function into a real linear regression model for sales forecasting.

I'll give you a minimal example using scikit-learn, which:

1. Reads a dataset (CSV/Excel)
2. Trains a Linear Regression model
3. Predicts future sales

Here's the code:

```
import pandas as pd  
  
from sklearn.linear_model import LinearRegression  
  
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import mean_squared_error

# 1 Load dataset
# Replace with your real file path
df = pd.read_csv("sales_data.csv")

# Example columns: 'Marketing_Spend', 'Season_Index', 'Sales'
X = df[['Marketing_Spend', 'Season_Index']] # Features
y = df['Sales'] # Target

# 2 Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 3 Create and train the model
model = LinearRegression()
model.fit(X_train, y_train)

# 4 Make predictions
y_pred = model.predict(X_test)

# 5 Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")

# 6 Function to predict sales (replaces dummy function)
def predict_sales(features):
    """
    Predict sales based on input features.
    """

Predict sales based on input features.
```

Args:

features (list or array): Numeric values in the same order as the training features.

Returns:

float: Predicted sales.

"""

```
return model.predict([features])[0]
```

# Example usage

```
print(predict_sales([2500, 4])) # Example: marketing spend, season index
```

How it works now:

- Instead of hardcoding a made-up formula like in your dummy function, the ML model learns  $\beta_0$  (intercept) and  $\beta_1, \beta_2, \dots$  (weights) from the training data.
- When you call `predict_sales(...)`, it uses the trained coefficients to give a real prediction.

If you want, I can adapt this exactly to your ASX dataset columns so it predicts something like *next month's sales from your stock or company financial data*.