IT496: Introduction to Data Mining

Lecture 14

Linear Regression

[Univariate and Multivariate Regression]

Arpit Rana

14th September 2023

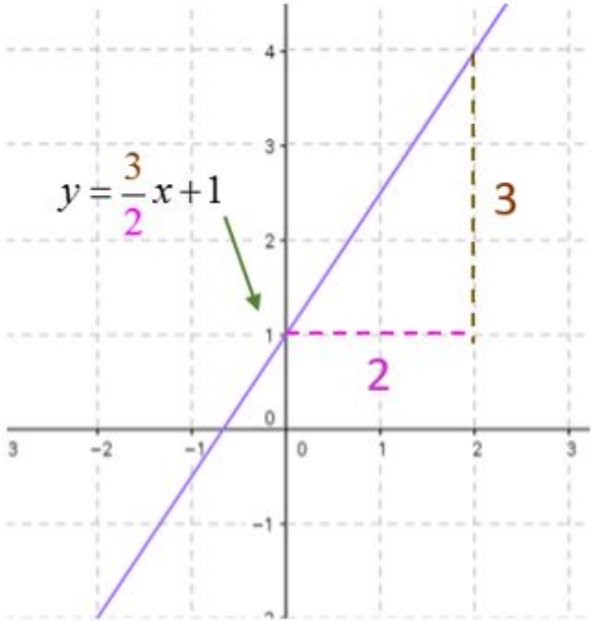
Linear Equations

We know the equation of a straight line:

$$y = mx + c$$

Gradient of the line (can be positive/ negative)

Intercept of the line (can be positive/ negative)



Linear Equations and Vectors

In general:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

- $\beta_0, \ldots, \beta_n$ are numbers, called the **coefficients**;
- $x_1, \ldots, x_n$ are the **variables**;
- Each of the things being added together is called a **term**.

Given a linear equation and the values of the variables $(x_1, \ldots, x_n)$, we can evaluate the equation, i.e. work out the value of y.

Linear Equations and Vectors

Given a general equation:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

- We can gather the variables into a *row vector* $[x_1\ x_2 \ldots x_n]$
- We can gather the coefficients (except β_0) into a *column vector* $\begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$ (of the same dimension, n)
- From an equation $y = 12 + 3x_1 + 4x_2 + 5x_3$ we get $x = [x_1\ x_2\ x_3]$ and $\beta = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}$

Linear Equations and Vectors

Given a general equation

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + . \ . \ . + \beta_n x_n$$

- Hence, the above equation can be written as

$$y = \beta_0 + \sum_{i=1}^{n} x_i \beta_i$$

- It can also, equivalently, be written in this form

$$y = \beta_0 + x\beta$$

Hence, to evaluate a linear equation, simply multiply the two vectors and add β_0

Linear Equations and Vectors

Given a general equation

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + . \ . \ . + \beta_n x_n$$

- Hence, the above equation can be written as

$$y = \beta_0 + \sum_{i=1}^{n} x_i \beta_i$$

- In a more simplified manner

$$y = \sum_{i=0}^{n} x_i \beta_i = x\beta$$ (here, $x_0 = 1$)

This is *vectorization* form: concise and fast code!

House Rent Prediction Dataset (Magicbricks, India)

	Posted On	BHK	Rent	Size	Floor	Area Type	Area Locality	City	Furnishing Status	Tenant Preferred	Bathroom	Point of Contact
311	2022-06-03	1	9000	450	Ground out of 3	Carpet Area	Salt Lake City Sector 5	Kolkata	Unfurnished	Bachelors/Family	1	Contact Agent
3869	2022-05-20	3	19500	1270	1 out of 2	Super Area	Madipakkam	Chennai	Semi-Furnished	Bachelors	2	Contact Owner
1368	2022-06-21	1	20000	310	Ground out of 7	Carpet Area	Malad West	Mumbai	Unfurnished	Bachelors	1	Contact Agent
1528	2022-06-13	2	16000	600	1 out of 2	Carpet Area	Girinagar	Bangalore	Unfurnished	Bachelors	2	Contact Owner
309	2022-06-25	3	13000	950	Ground out of 2	Carpet Area	Rabindrapally, Garia	Kolkata	Unfurnished	Bachelors/Family	2	Contact Owner

Dataset Glossary (Column-Wise): 4746 Records

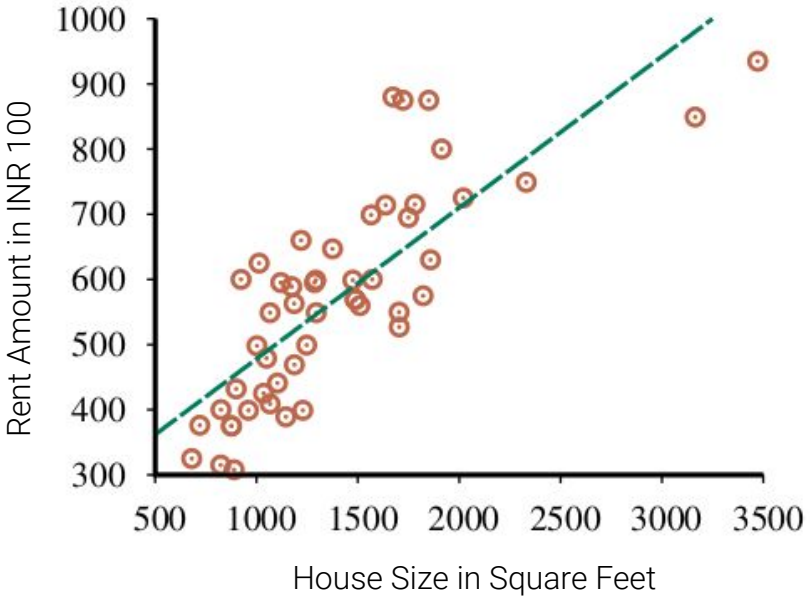
- **BHK**: Number of Bedrooms, Hall, Kitchen.
- **Floor**: Ground out of 2, 3 out of 5, etc.
- **Size**: Size of property in Square Feet.
- **Area Type**: Super Area/Carpet Area/Built Area.
- **Furnishing Status**: Furnished/Semi-Furnished/ Unfurnished.
- **Bathroom**: Number of Bathrooms.
- **Area Locality:** Locality of the property
- **City**: City where the property is Located.
- **Tenant Preferred**: Family/Bachelor
- **Point of Contact**: Agent / Owner
- **Rent**: Price of the property

Univariate Linear Regression (with one variable)

The goal of our learning algorithm is to fit a linear model to this data:

$$\hat{y} = \beta_0 + \beta_1 \times size$$

In other words, our goal is to choose values for β_0 and β_1

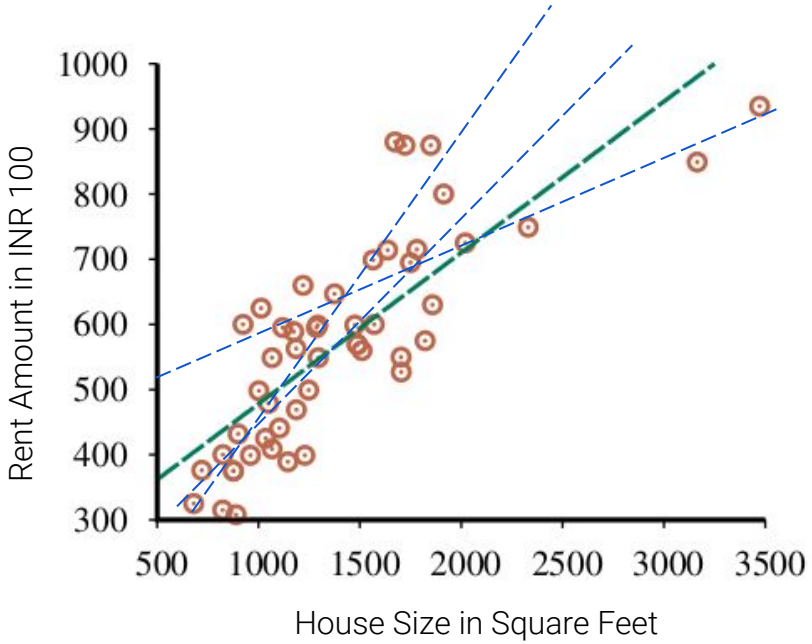


Univariate Linear Regression (with one variable)

But there is an infinite set of linear models the algorithm can choose from:

- an infinite number of straight lines it can draw;
- or, equivalently, an infinite set of values from which it can pick β_0 and β_1

We want it to choose the one that best fits the data.



Univariate Linear Regression (with one variable)

The algorithm needs a function that measures how well a model (hypothesis) fits the data.

- This is called its **loss function**, designated as J.
- The function takes in a particular hypothesis h_β and gives it a score.
  - Low numbers are better!
- For each $\boldsymbol{x}$ in the training set, it will compare $h_\beta(\boldsymbol{x})$, which is the prediction that h_β makes on $\boldsymbol{x}$, with the actual value y.

The loss function most usually used for linear regression is the ***mean squared error***, i.e.:

$$J(X, y, \beta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\beta\left(x^{(i)}\right) - y^{(i)} \right)^2$$
$$= \frac{1}{2} mean(X\beta - y)^2$$

This is often referred to as *ordinary least-squares* regression (OLS).

Univariate Linear Regression (with one variable)

The goal of our learning algorithm is to fit a linear model to this data:

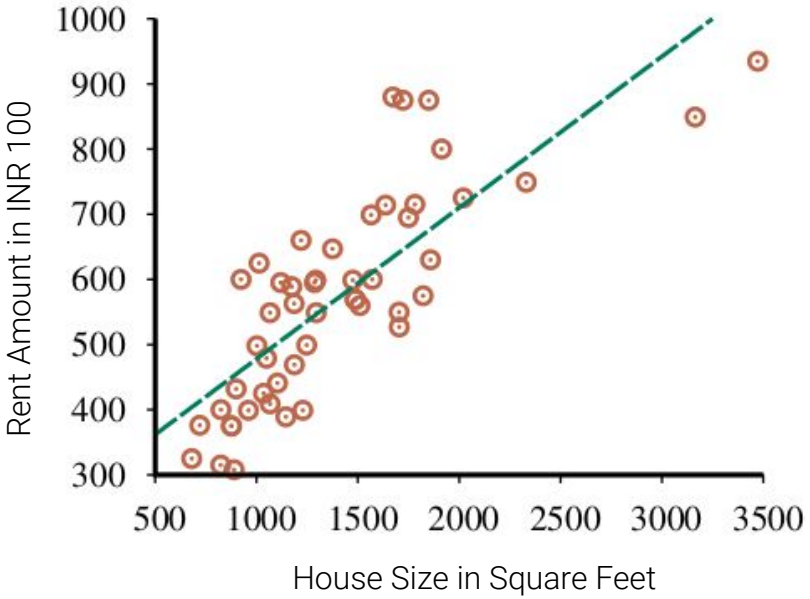
$$\hat{y} = \beta_0 + \beta_1 \times size$$

We found the solution is -

β_0 = 0.232, and β_1 = 246, i.e.

the line y = 0.232x + 246

minimizes the loss function.



Multivariate Linear Regression (with more than one variable)

We can simply generalize the idea to multiple variables:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

Equivalently -

$$y = \sum_{i=0}^{n} x_i \beta_i = x\beta$$

The only differences when we move to more than one feature:

- We can't plot so easily.
- The model is a plane when there are two features.
- The model is a hyperplane when there are more than two features.

Finding OLS Models

- We've been trying out different values for β, looking for the model with lowest mean squared error...
  - ...by trial and error!

  In practice, it is not done by trial-and-error.

- There are two main methods:
  - The **Normal Equation** (`LinearRegression` class in scikit-learn)
  - Various forms of **Gradient Descent** (`SGDRegressor` class in scikit-learn)

The Normal Equation

- The normal equation solves for β

$$\beta = \left(X^T X\right)^{-1} X^T y$$

  i.e., the normal equation gives us the parameters that minimize the loss function.

- Where does it come from?
  - Take the gradient of the loss function: $\frac{1}{m} X^T (X\beta - y)$
  - Set it to zero: $\frac{1}{m} X^T (X\beta - y) = 0$ (in fact, a (n+1)-dimensional vector of zeros).
  - Then do some algebraic manipulation to get β on the left-hand side, that's it.

The Normal Equation: Partial Derivatives

We need the gradient of the loss function with regards to each β_j

- In other words, how much the loss will change if we change β_j a little.
- With respect to a particular β_j, it is called the partial derivative
- The partial derivatives of $J(X, y, \beta)$ with respect to β_j, are

$$\frac{\partial J(X, y, \beta)}{\partial \beta_j} = \frac{1}{m} \sum_{i=1}^{m} \left(x^{(i)} \beta_j - y^{(i)} \right) \times x_j^{(i)}$$

The Normal Equation: Partial Derivatives

- The gradient vector, $\nabla_\beta J(X, y, \beta)$ is a vector of each partial derivative:

$$\nabla_\beta J(X, y, \beta) = \begin{bmatrix} \frac{\partial J(X,y,\beta)}{\partial \beta_0} \\ \frac{\partial J(X,y,\beta)}{\partial \beta_1} \\ \cdot \\ \cdot \\ \cdot \\ \frac{\partial J(X,y,\beta)}{\partial \beta_n} \end{bmatrix} = \begin{bmatrix} \frac{1}{m} \sum_{i=1}^{m} x^{\cdot(i)} \beta_j - y^{(i)} \times x_0^{\cdot(i)} \\ \frac{1}{m} \sum_{i=1}^{m} x^{\cdot(i)} \beta_j - y^{(i)} \times x_1^{\cdot(i)} \\ \cdot \\ \cdot \\ \cdot \\ \frac{1}{m} \sum_{i=1}^{m} x^{\cdot(i)} \beta_j - y^{(i)} \times x_n^{\cdot(i)} \end{bmatrix}$$

- Also, there is a vectorized way to compute it:

$$\nabla_\beta J(X, y, \beta) = \frac{1}{m} X^T (X\beta - y)$$

Multivariate Linear Regression (with more than one variable)

Python Implementation

The `fit` method of scikit-learn's `LinearRegression` class does what we have described:

- It inserts the extra column of 1s.
- It calculates β using the Normal Equation.

There's a problem:

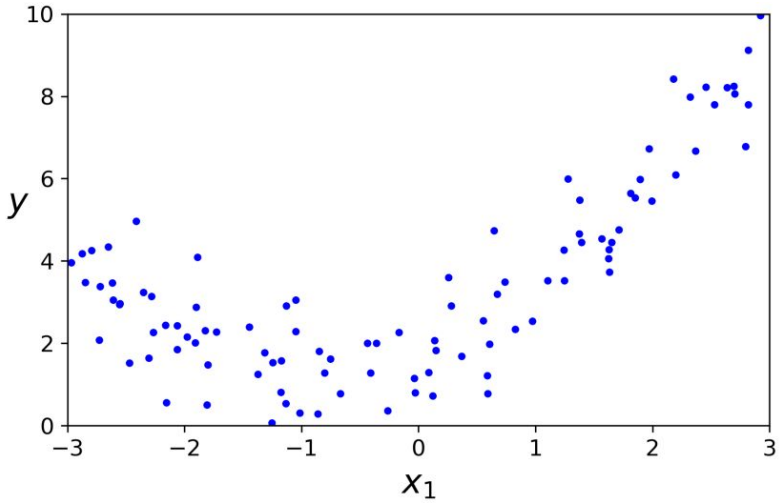
- The normal equation requires that $\mathbf{X}^T\mathbf{X}$ has an inverse.
- But it might not (might be a singular matrix).
- Therefore, we can use the pseudo-inverse instead. (in place of `numpy.linalg.inv()`, we can use `numpy.linalg.pinv()`).

Non-linearity

What if the true relationship between the features and the target values is **non-linear**?

A linear model may not be good enough:

- The test error may be too high
- Even the training error may be too high!



Polynomial Regression

What we need is a more complex model.

- Roughly, a model is more complex if it has more parameters.
  - E.g. quadratic functions for a dataset with just three features ($\boldsymbol{x}_1$, $\boldsymbol{x}_2$ and $\boldsymbol{x}_3$)

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_1^2 + \beta_5 x_2^2 + \beta_6 x_3^2 + \beta_7 x_1 x_2 + \beta_8 x_2 x_3 + \beta_9 x_3 x_1$$

Polynomial Regression

There are learning algorithms for non-linear models (including neural networks, see later).

But there's a really neat trick for using a **linear model** to get some of the same effect!

- We add extra features to our dataset.
    - Some of the new features will be powers of the original features,
      e.g. a new feature $x_4 = x_1^2$
    - Others will be products of the original features,
      e.g. a new feature $x_7 = x_1 x_2$ (these are often called **interaction features**).
- Then learn a linear model on the new dataset.

This technique is called **Polynomial Regression**.

Polynomial Regression

Example:

- We want to learn models of this form:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_1^2 + \beta_5 x_2^2 + \beta_6 x_3^2 + \beta_7 x_1 x_2 + \beta_8 x_2 x_3 + \beta_9 x_3 x_1$$

- But instead we learn linear models of this form:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5 + \beta_6 x_6 + \beta_7 x_7 + \beta_8 x_8 + \beta_9 x_9$$

- but where

$$\begin{aligned} x_4 &= x_1^2 \\ x_5 &= x_2^2 \\ x_6 &= x_3^2 \\ x_7 &= x_1 x_2 \\ x_8 &= x_2 x_3 \\ x_9 &= x_3 x_1 \end{aligned}$$

Polynomial Regression

Polynomial Regression in scikit-learn

- There's a class called `PolynomialFeatures`
- We use it as follows

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
```

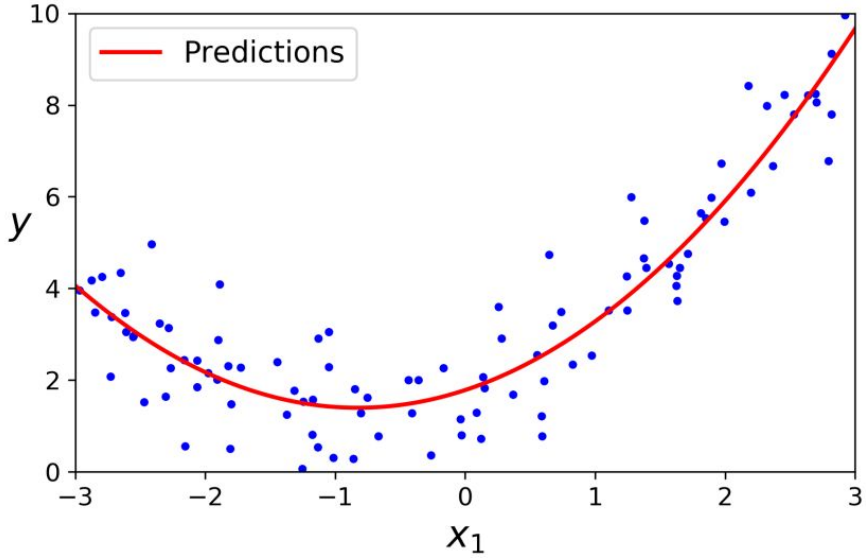
- `PolynomialFeatures(degree=d, include_bias=False)` transforms a dataset that had n features into one that has $(n+d)!/n!d!$ features.

Polynomial Regression

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

$$\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$$

$$y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise.}$$



Next lecture

Regularization

15th September 2023