Date

# Data Preprocessing : One Hot Encoding

[creating new features
- feature engineering]

Suppose a data set that consists of four columns.

1. locality - text data
2. area_sq_ft - number
3. bedrooms - 4
4. price_lakhs - n

property price depends upon locality also.

computer understands numbers.
text can be converted into numbers - feature engineering

Method 1 - Label Encoding

Assign numbers to text data

eg:

Locality (categorical data)
Kollur - 1
Banjarahills - 2
Mankhal - 3

categorical data

1. ~~ordinal~~ nominal (no order).
2. ~~nominal~~ ordinal (categories have some kind of order)

eg: green, red, blue.

Kollur <
Banjarahills <
Mankhal

One hot encoding is used.

bachelors < masters < phd
dissatisfied < neutral < satisfied
low, medium, high

Label encoding is used

| A Locality | B locality | C locality | Area sq-ft | Bed rooms | Price lakhs |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 656 | 3 | 39 |
| 1 | 0 | 0 | 1260 | 2 | 83 |
| 1 | 0 | 0 | 1057 | 3 | 86 |
| 0 | 1 | 0 | 2400 | 3 | 300 |
| 0 | 1 | 0 | 1008 | 2 | 50 |
| 0 | 0 | 1 | 1125 | 2 | 64 |
| 0 | 0 | 1 | 1700 | 3 | 100 |

Locality A, Locality B, Locality C – Dummy variables

Original column – locality

On that column we apply onehot encoding and we get 3 dummy columns

In this one of the column is redundant.

If we know two set of columns, from that we can infer the third one.

If it is neither A or B then it will be C

Multicollinearity:- Occurs when two or more independent variables are highly corelated, making it difficult to distinguish their individual effects on the dependent variable.

To overcome this problem, ~~are~~ drop any one column.

pd.get_dummies(df, columns = ["locality"])

due to multicolinearity column one col is dropped.

pd.get_dummies(df, columns = ["locality"],
drop_first = True)
↳ drop 1st dummy variable

df_encoded = pd.get_dummies(df, columns =
["locality"], drop_first = True)

df_encoded.sample(5)

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

X = df_encoded.drop('price_lakhs', axis=1)
y = df_encoded['price_lakhs']

X_train, X_test, y_train, y_test = train_test_split(
X, y, test_size = 0.2, random_state = 42)

model = LinearRegression
model.fit(X_train, y_train)

model.score(X_test, y_test)

test = pd.DataFrame([
{'area_sq_ft': 1600, 'bedrooms': 2,
"locality_kollur": False, "locality_
Mankhal": False}
])

Date ___ / ___ / ___

model.predict(test)

Date ___ / ___ / _____

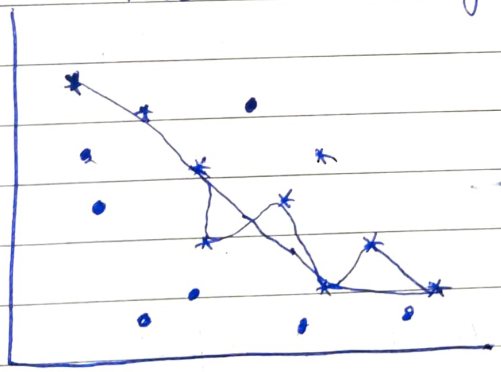
**Overfitting :-** Occurs when a model learns too much detail and noise from the training data, affecting its performance on new data.

**Underfitting :-** Happens when a model is too simple and can't learn the data pattern, leading to poor performance on all data.

**Balanced fit** :- Achieved when a model accurately ~~known~~ learns the training data patterns and performs well on unseen data.

Overfitting

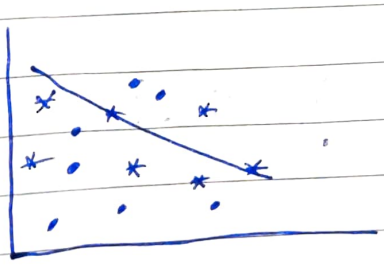
- Train error - low
  Test error - High



• - test datapoints
* - train "

learning from noise/ fluctuations not trying to learn general patterns

underfit

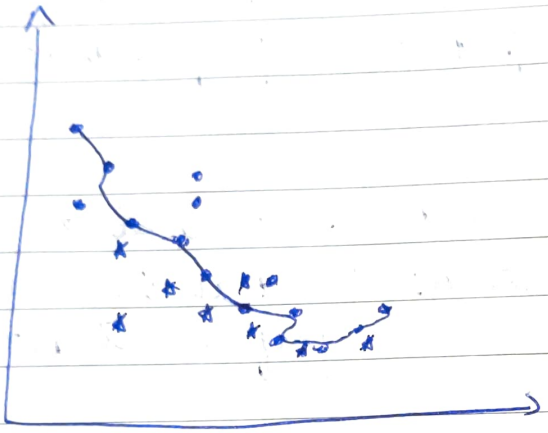


test & train error - high

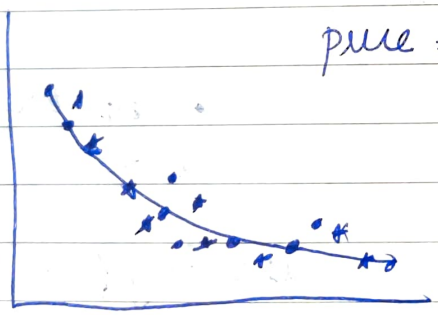
Regularization
→ used to reduce overfitting

Let an overfit model using polynomial regression

$price = \beta_0 + \beta_1 * mileage + \beta_2 * mileage^2 + \beta_3 * mileage^3 + \beta_4 * mileage^4 + \beta_5 * mileage^5$



right model here is this



$price = \beta_0 + \beta_1 * mileage + \beta_2 * mileage^2$

If we try to reduce $\beta_3$, $\beta_4$ & $\beta_5$ as zero we will get second eqn.

$$mse = \frac{1}{n} \sum_{i=1}^{n} (y_i - y_{predicted})^2$$

Date ___/___/___

here ypredicted is

$$price = \beta_0 + \beta_1 * mileage + \beta_2 * mileage^2 + \beta_3 * mileage^3 + \beta_4 * mileage^4 + \beta_5 * mile$$

In general

$$y_{predicted} = \beta_0 + \beta_1 x + \beta_2 * x^2 + \dots \beta_n * x^n$$

We need to reduce $\beta_3$, $\beta_4$ & $\beta_5$ close to zero so that we will get a balanced fit model.

for that we are adding a new term to the equation.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - y_{predicted})^2 + \lambda \sum_{i=1}^{n} \beta_i^2$$

$\lambda \rightarrow$ a hyperparameter that we can configure

$$\sum_{i=1}^{n} \beta_i^2 = \beta_0^2 + \beta_1^2 + \beta_2^2 - \dots$$

If larger $\beta_3$ value is there let it be 100

$100 \times 100 * 100 = 10000$

a big number.

$\beta_3 * x^3$ gives again a big no.

If we substitute this value in equation for y predicted MSE will also increases

so Gradient Descent try to reduce the MSE

So it will configure $\beta_3$ or $\beta_5$ value

Thus we get a balanced fit
(all large polynomial values → 0)
This is known as $L_2$ regularization
This is also called ridge regression

$L_1$ regularization / Lasso regression

penalty term adding is $|\beta_i|$
absolute value of $\beta$

$$mse = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_{predicted})^2 + \sum_{i=1}^{n}|\beta_i|$$

$$y_{predicted} = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \ldots \beta_n x^n$$

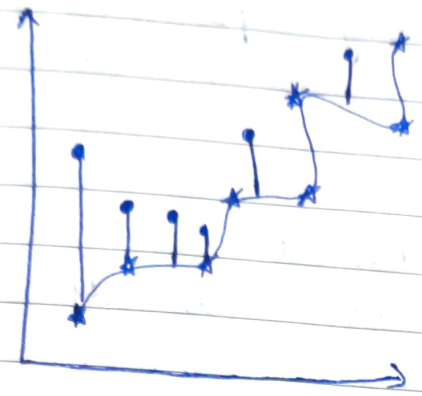
Let $\beta_3 = 100$

$100 * x^3$ large value

MSE increases
GD tries to minimize error.
to minimize error
it minimize highest polynomial $\beta$ terms.

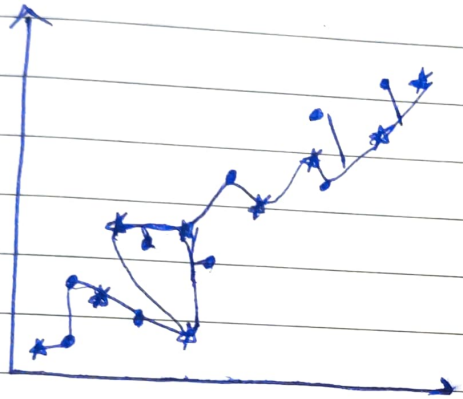
# Bias Variance Tradeoff

Eg:



Train dataset error = 0
Test " " = 100
higher test error

high variance in terms of test data



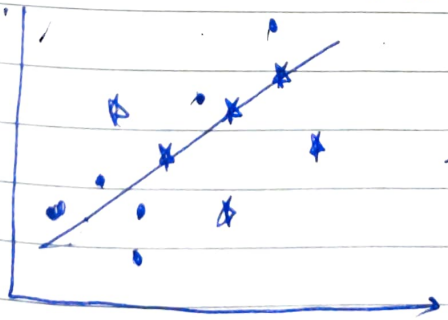
Train dataset error = 0
Test " " = 27
lower test error

overfitting

same data set.
test error varies

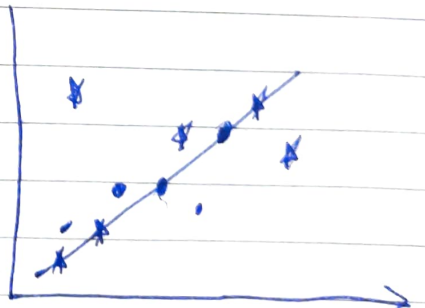
usually overfit model have high variance
Based on the selection of the training dataset, the prediction errors vary greatly (ie showing high variance among errors)

⊖ underfitting



high bias

error high ⇒ high bias



Test error = 47
Train " = 43

Test error = 37
Train error = 41

Bias is a measurement of how accurately a model can capture a pattern in a training dataset.

Date

In above case since error is big it is said to have high bias
When you try to built a very simple model, you will get high errors in testing & training data ⇒ high bias

In case of balanced data set
⇒ low variance, low bias.