Polynomial Regression

`from sklearn.preprocessing import PolynomialFeatures`

- Polynomial regression is a form of regression analysis used to model nonlinear relationships between a dependent variable y and an independent variable x.
- It extends linear regression by fitting a polynomial equation of degree n, allowing for more flexibility in capturing curvilinear trends.

Linear Regression: ***y=*** β_0**+**$\beta_1 x$**+**$\boldsymbol{\epsilon}$

Polynomial Regression: ***y=*** $\beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_n x^n + \epsilon$

Why Use Polynomial Regression?

- To model relationships where the effect of x on *y* is nonlinear.
- Example: The relationship between temperature (x) and electricity usage (y) might follow a quadratic curve.
- **Flexibility**: Captures complex patterns beyond straight lines.
- **Simplicity**: Still uses linear regression techniques (e.g., OLS) but with transformed features

Best Practices

- **Start with Low Degrees**: Begin with degree=2 or 3, then increase if needed.
- **Visualize the Fit**: Plot residuals to check for patterns (e.g., funnel shape → heteroscedasticity).
- **Compare Models**: Use AIC/BIC or cross-validation to compare polynomial and linear models.

Limitations

- **Interpretability**: Coefficients for x^n terms are harder to explain.
- **Extrapolation Risk**: Polynomial models often perform poorly outside the training data range.
- **Sensitivity to Outliers**: High-degree terms amplify the impact of outliers.

Real-World Example

Scenario: Predicting house prices (*y*) based on square footage (*x*).

Observation: The price increase slows down for very large houses.

Solution: Use a quadratic term (x^2) to model the diminishing returns.

Math

- For every value in x, you calculate x^1, x^2, x^3, etc.
  - Begin with degree=2 or 3
- This will have a polynomial shape
- Degree is a hyperparameter.
- Less value = Underfit
- High value = Overfit

Code

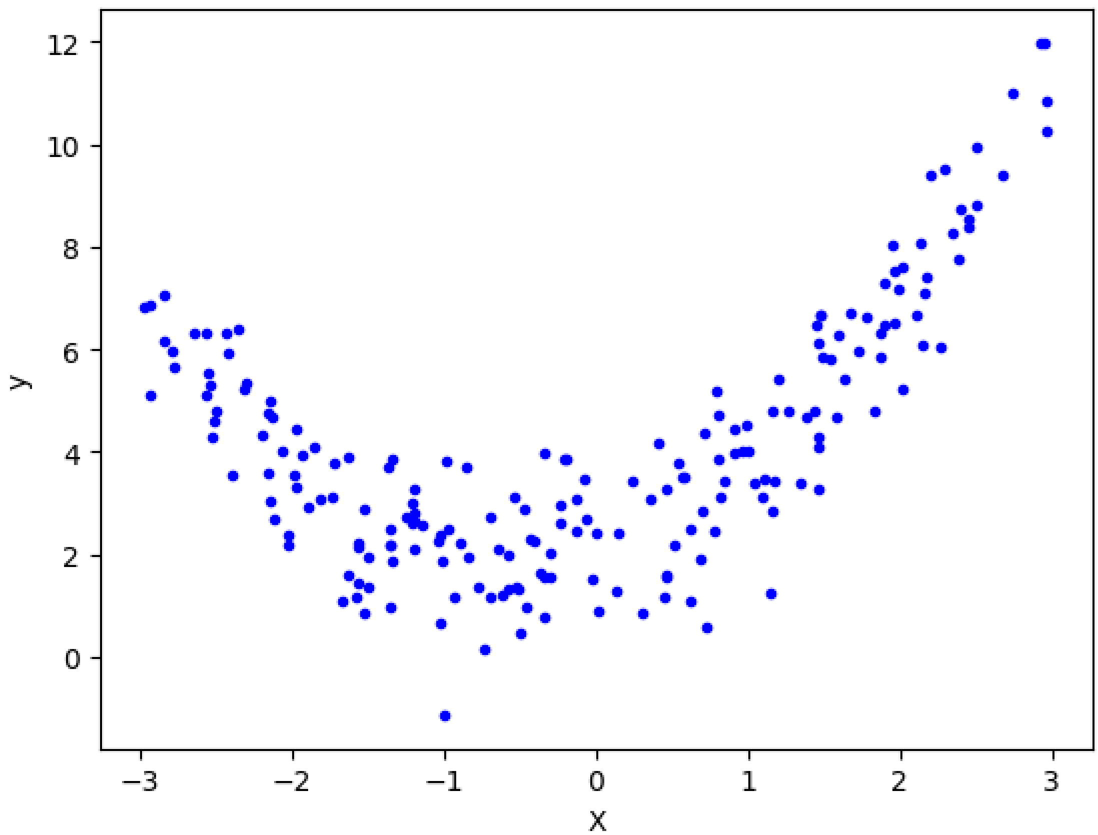
```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
```

```
from sklearn.linear_model import LinearRegression,SGDRegressor

from sklearn.preprocessing import PolynomialFeatures,StandardScaler

from sklearn.metrics import r2_score

from sklearn.pipeline import Pipeline
```

```
X = 6 * np.random.rand(200, 1) - 3
y = 0.8 * X**2 + 0.9 * X + 2 + np.random.randn(200, 1)

# y = 0.8x^2 + 0.9x + 2
```

```
plt.plot(X, y,'b.')
plt.xlabel("X")
plt.ylabel("y")
plt.show()
```

- `'b.'` → blue colour + `.` converts line into dots

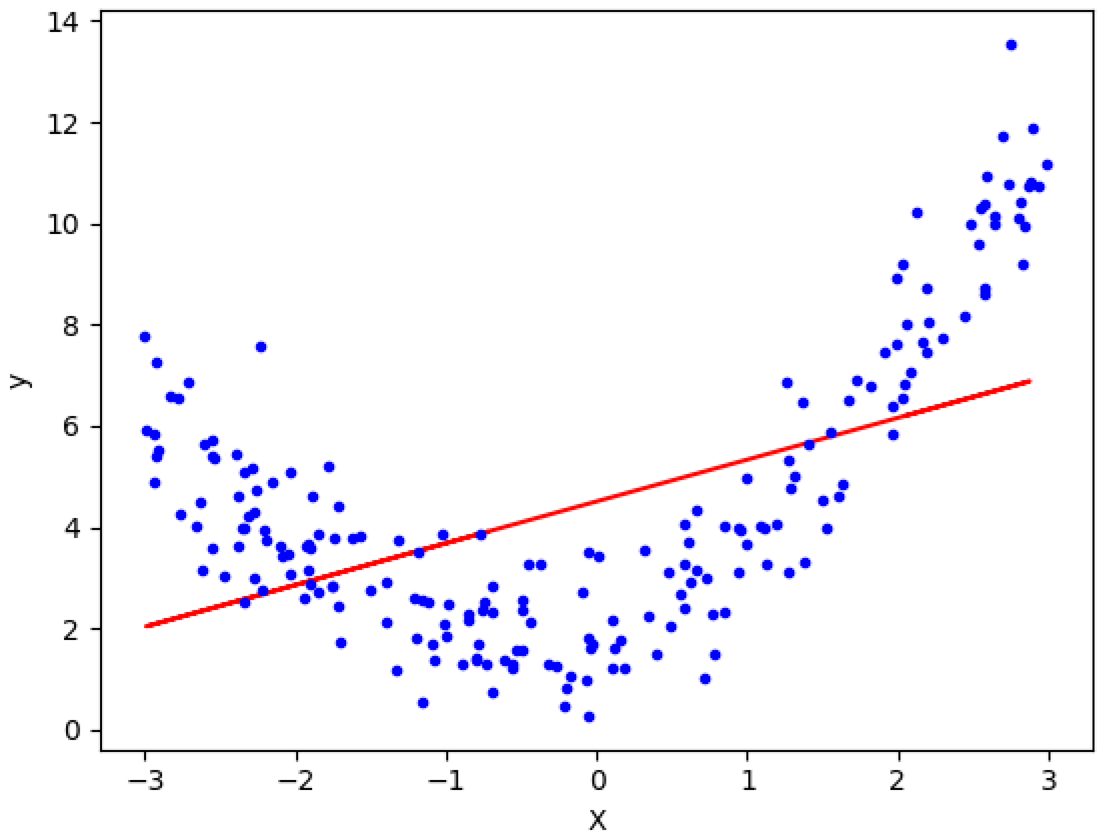


Try to apply LR on this data:

```
# Train test split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=
2)

# Applying linear regression
lr = LinearRegression()

lr.fit(X_train,y_train)

y_pred = lr.predict(X_test)
r2_score(y_test,y_pred)
```

Output: *0.2778958989749396*

```
plt.plot(X_train,lr.predict(X_train),color='r')
plt.plot(X, y, "b.")
plt.xlabel("X")
plt.ylabel("y")
plt.show()
```



- Line does not fit best

Now, Apply Polynomial Linear Regression

- First, we need to transform our features

Use → `from sklearn.` `preprocessing` `import` `PolynomialFeatures,StandardScaler`

- You have to define **degree**
- If degree=2, for every input column, you'll get 3 columns
  - x becomes → x^0, x^1, x^2

```
# Applying Polynomial Linear Regression
# degree 2
poly = PolynomialFeatures(degree=2,include_bias=True)

X_train_trans = poly.fit_transform(X_train)
X_test_trans = poly.transform(X_test)
```

- `include_bias=True` : default is true only
  - If you make it false → You won't get x^0
  - You have to try both True & False and test the model
- `interaction_only` **: *default=False***
  - you **only get features that are products of *different* input features.**
  - `interaction_only=` `False` **:** You'd get features like: `a`, `b`, `c`, `a²`, `b²`, `c²`, `ab`, `ac`, `bc`, `a³`, `b³`, `c³`, `a²b`, etc. (all combinations and powers up to a certain degree).
  - `interaction_only=` `True` **:** You'd **only** get interaction terms (products of different features) and the original features themselves, but **no powers** of individual features. For degree 2 with `interaction_only`, you'd get: `a`, `b`, `c`, `ab`, `ac`, `bc`.
  - You would **not** get `a²`, `b²`, `c²`.

Ex. of `PolynomialFeatures`

	A	B	C
0	1	4	0
1	2	4	10
2	3	4	100

```
# Output columns: A, B, C, A*B, A*C, B*C
poly.fit_transform(X)
```

```
array([[  1.,   4.,   0.,   4.,   0.,   0.],
       [  2.,   4.,  10.,   8.,  20.,  40.],
       [  3.,   4., 100.,  12., 300., 400.]])
```

```
print(X_train[0])
print(X_train_trans[0])

Output:
[2.03304836] ← Original data
[1.         2.03304836 4.13328563] ⟵ x^0, x^1, x^2
```

- Now train the model again with `X_train_trans` & `y_train`

```
lr = LinearRegression()
lr.fit(X_train_trans,y_train)
```

- Instead of x_train, we passed X_train_trans

```
y_pred = lr.predict(X_test_trans)
r2_score(y_test,y_pred)
```

Output: ***0.8372014016382803***

- The previous r2 score was 0.2778958989749396

```
print(lr.coef_)
print(lr.intercept_)

Output:
[[0.         0.87356415 0.8484806 ]] ← x^0, x^1, x^2
[1.8369744] ← Intercept
```

In our equation → $y = 0.8x^2 + 0.9x + 2$

x^1 → 0.9

x^2 → 0.8

Intercept → 2

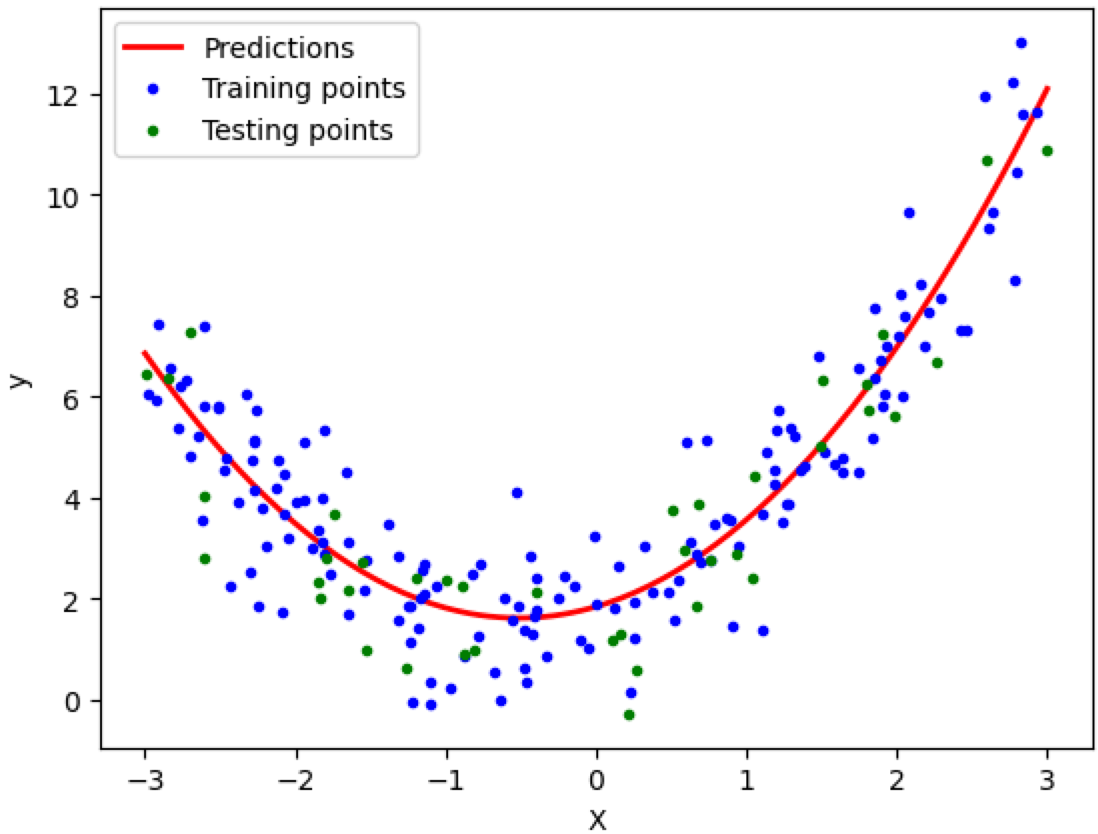
- The predicted values by the model are very close to our original equation
- It's not perfect due to random noise that we added with `np.random.randn(200, 1)`

```
X_new=np.linspace(-3, 3, 200).reshape(200, 1)
X_new_poly = poly.transform(X_new)
y_new = lr.predict(X_new_poly)
```

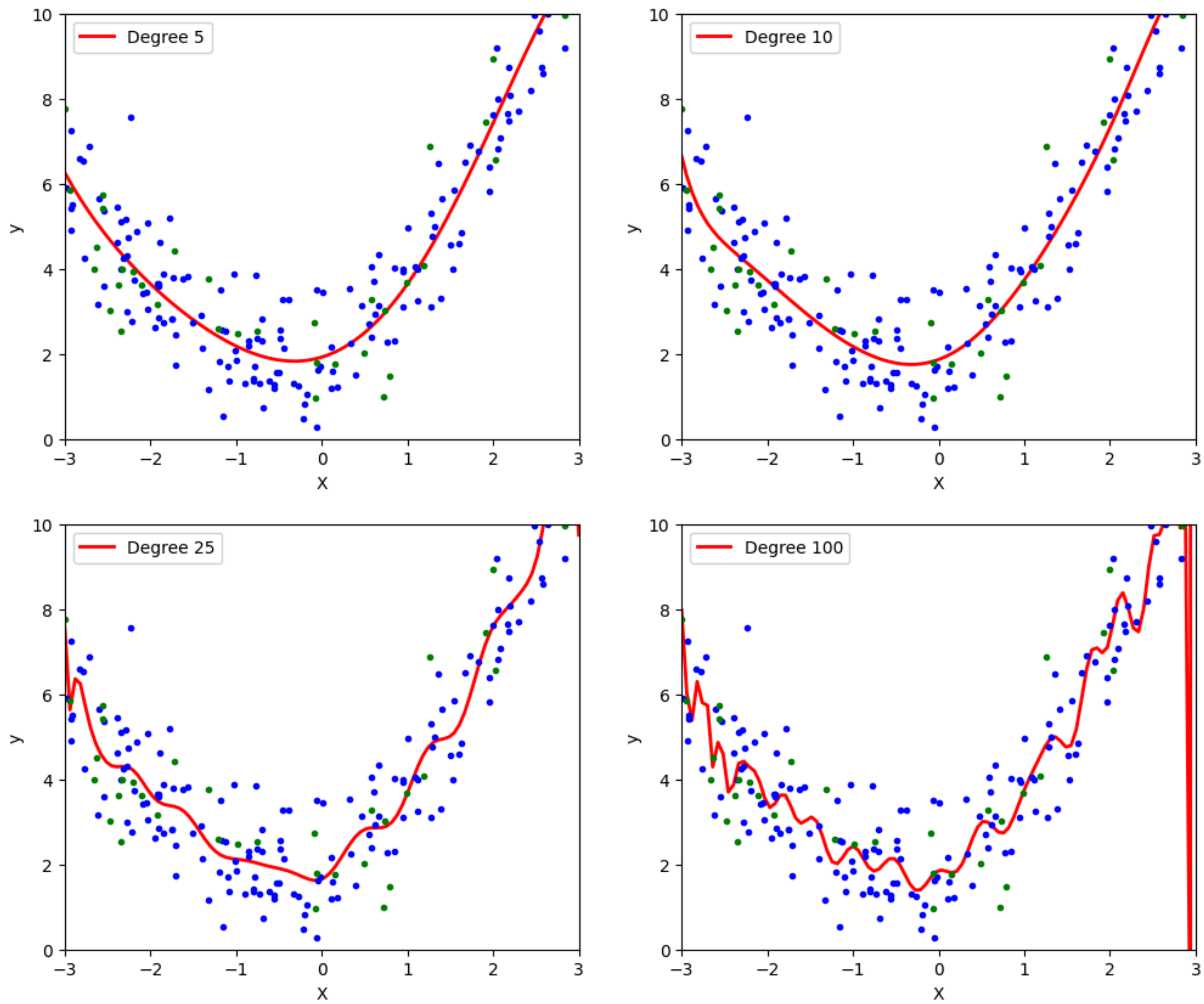
1. We generated 200 linearly spaced points between -3 & 3 with `np.linspace(-3, 3, 200).reshape(200, 1)`
2. Transform the above data with → `X_new_poly = poly.transform(X_new)`
3. Predict the Y for these newly generated values → `lr.predict(X_new_poly)`

```
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.plot(X_train, y_train, "b.",label='Training points')
```

```
plt.plot(X_test, y_test, "g.",label='Testing points')
plt.xlabel("X")
plt.ylabel("y")
plt.legend()
plt.show()
```



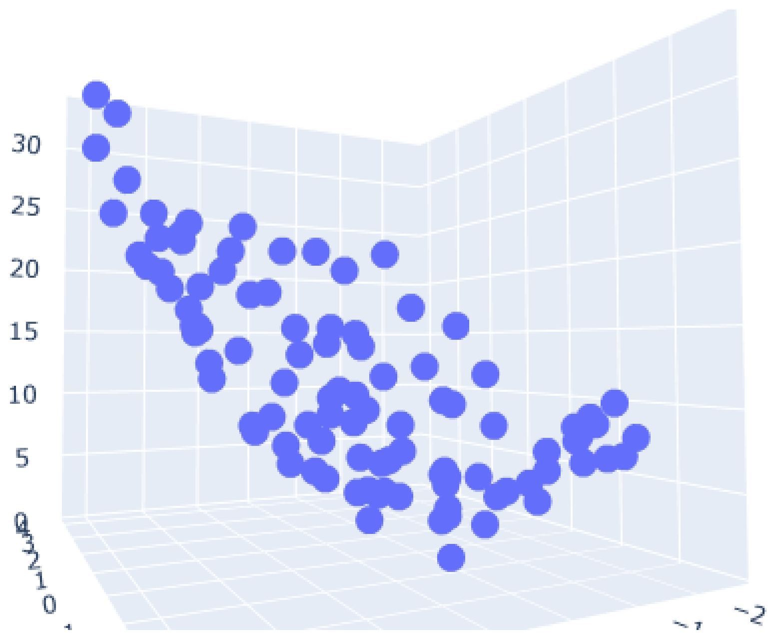
Effect of Degrees



More than 1 Input column

```
# 3D polynomial regression
x = 7 * np.random.rand(100, 1) - 2.8
y = 7 * np.random.rand(100, 1) - 2.8
z = x**2 + y**2 + 0.2*x + 0.2*y + 0.1*x*y +2 + np.random.randn(100, 1)
# z = x^2 + y^2 + 0.2x + 0.2y + 0.1xy + 2
```

```
import plotly.express as px
df = px.data.iris()
fig = px.scatter_3d(df, x=x.ravel(), y=y.ravel(), z=z.ravel())
fig.show()
```

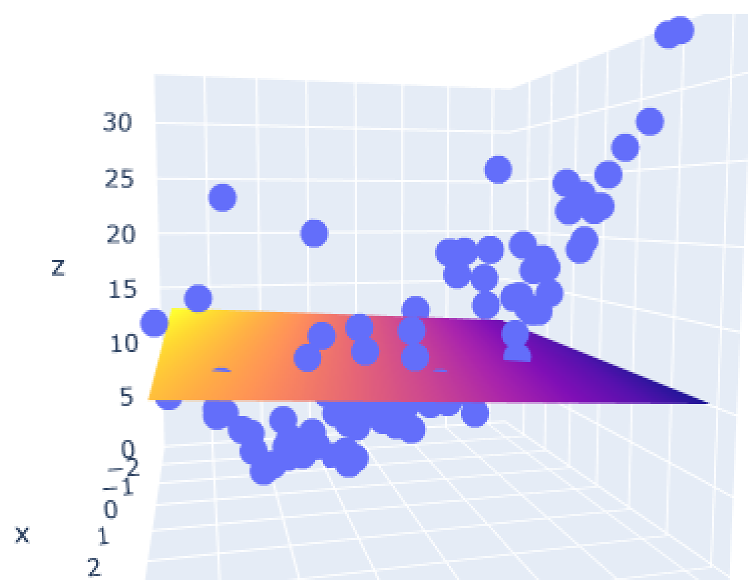


- We'll try to apply simple LR to this data

```
lr = LinearRegression()
lr.fit(np.array([x,y]).reshape(100,2),z)

x_input = np.linspace(x.min(), x.max(), 10)
y_input = np.linspace(y.min(), y.max(), 10)
xGrid, yGrid = np.meshgrid(x_input,y_input)

final = np.vstack((xGrid.ravel().reshape(1,100),yGrid.ravel().reshape(1,100))).T
```

```
z_final = lr.predict(final).reshape(10,10)
```

```
import plotly.graph_objects as go

fig = px.scatter_3d(df, x=x.ravel(), y=y.ravel(), z=z.ravel())

fig.add_trace(go.Surface(x = x_input, y = y_input, z =z_final ))

fig.show()
```



- The output is 👆

- Now, we'll apply polynomial features to this

```
X_multi = np.array([x,y]).reshape(100,2)
X_multi.shape
```

```
Output: (100,2)
```

```
poly = PolynomialFeatures(degree=3)
X_multi_trans = poly.fit_transform(X_multi)
```

```
print("Input", poly.n_features_in_)
print("Output", poly.n_output_features_)
print("Powers\n", poly.powers_)
```

Output:

Input 2
Output 10
Powers
[[0 0]
[1 0]
[0 1]
[2 0]
[1 1]
[0 2]
[3 0]
[2 1]
[1 2]
[0 3]]

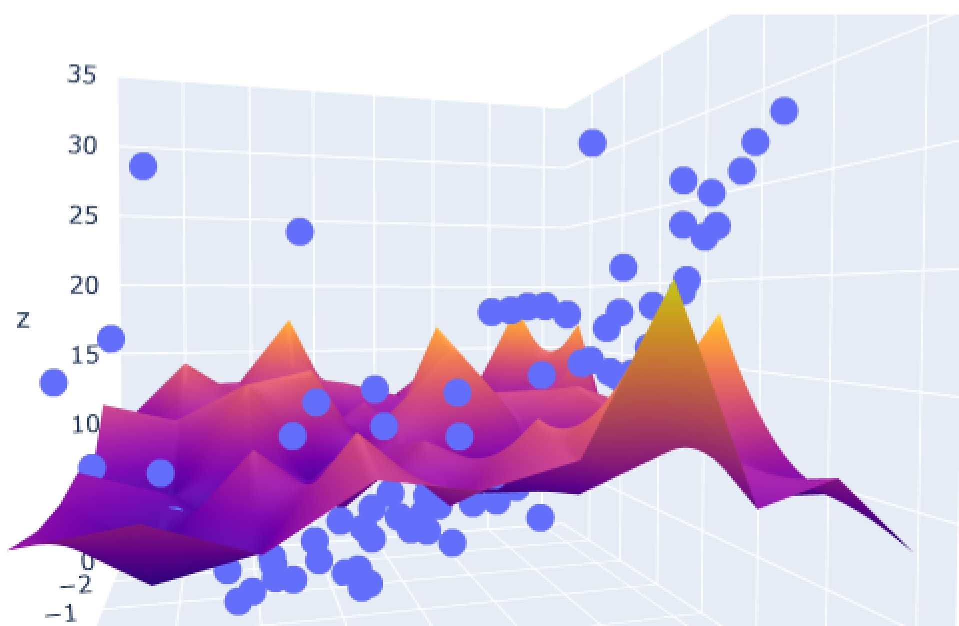
- Degree is 3. Therefore, sum of the 👆 terms we'll be 3 or less

```
X_multi_trans.shape
```

```
Output: (100, 10)
```

```
lr = LinearRegression()
lr.fit(X_multi_trans,z)
X_test_multi = poly.transform(final)
z_final = lr.predict(X_multi_trans).reshape(10,10)
```

```
fig = px.scatter_3d(x=x.ravel(), y=y.ravel(), z=z.ravel())

fig.add_trace(go.Surface(x = x_input, y = y_input, z =z_final))

fig.update_layout(scene = dict(zaxis = dict(range=[0,35])))

fig.show()
```



- With degree 2: