50.039 Theory and Practice of Deep Learning

W1-S2 Introduction and Machine Learning Reminders

Matthieu De Mari

About this week (Week 1)

1. What are the **typical concepts of Machine Learning** to be used as a starting point for this course?
2. What are the **different families of problems** in Deep Learning?
3. What is the **typical structure of a Deep Learning problem**?
4. What is **linear regression** and how to implement it?
5. What is the **gradient descent algorithm** and how is it used to **train Machine Learning models**?
6. What is **polynomial regression** and how to implement it?
7. What is **regularization** and how to implement it in **Ridge regression**?

About this week (Week 1)

8. What is **overfitting** and why is it bad?
9. What is **underfitting** and why is it bad?
10. What is **generalization** and how to evaluate it?
11. What is a **train-test split** and why is it related to **generalization**?
12. What is a **sigmoid** function? What is a **logistic** function?
13. How to perform **binary classification** using a **logistic regressor** and how is it related to linear regression?

About this week (Week 1)

14. What are **Neural Networks** and how do they relate to the **biology of a human brain**?
15. What is a **Neuron** in a Neural Network and how does it relate to linear/logistic regression?
16. What is the **difference** between a **shallow** and a **deep neural network**?
17. How to **implement a shallow Neural Network** manually and define a **forward propagation** method for it?
18. How to **train a shallow Neural Network** using **backpropagation**? How to define **backward propagation** and **trainer** functions?

A problem with the normal equation

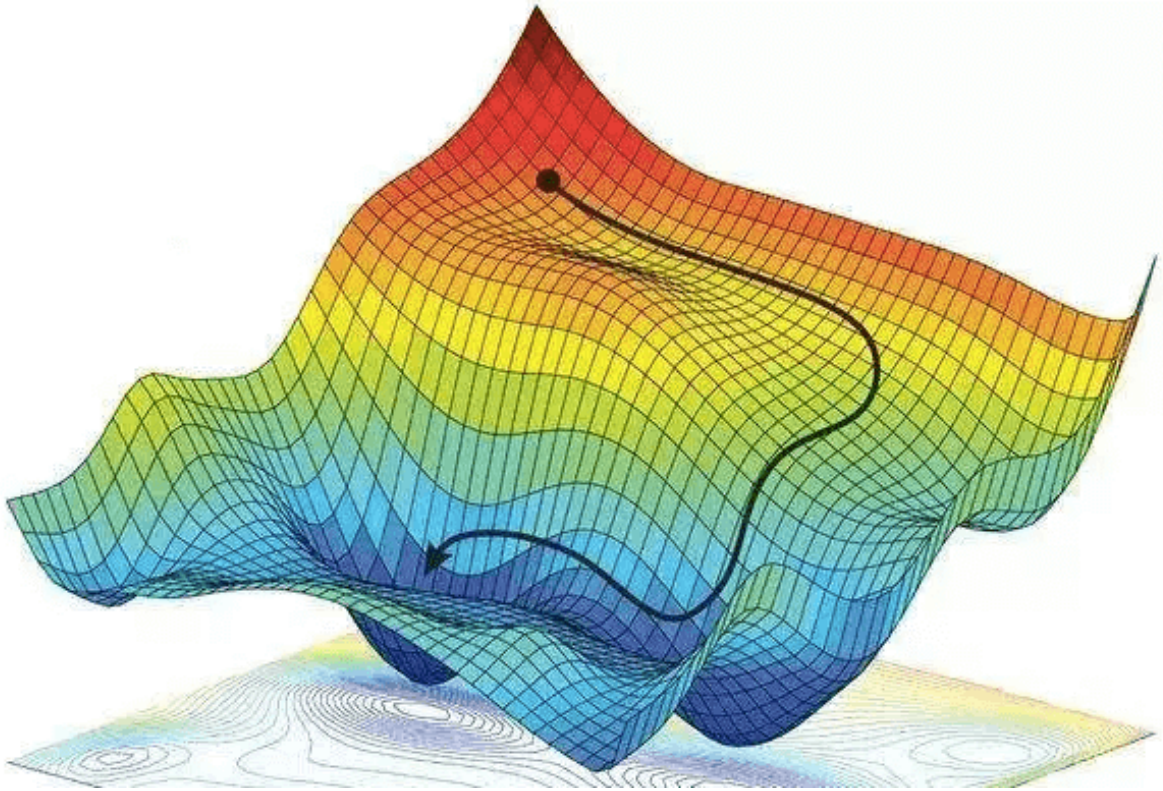
- The normal equation $W^* = (X^T X)^{-1} X^T Y$ immediately gives the optimal set of parameters (a^*, b^*) to use for linear regression.
- However, it can become **computationally expensive when the number of features is very large**.
- It might even be **impossible to use when the matrix of feature variables $(X^T X)^{-1}$ is not invertible**.
- **Problem: In these cases, the normal equation can be slow to compute, or it may not even have a solution.**
- **Another problem: More sophisticated models than linear regression will not have a normal equation, anyway.**

Gradient Descent, to the rescue!

Definition (Gradient Descent):

Gradient Descent (GD) is an iterative algorithm used to solve optimization problems.

- It starts with an initial set of parameters.
- It then repeatedly updates the parameters in the direction of the negative gradient of the given cost function, until it converges to a local minimum.



Forgot about GD?
Have a look at your notes from 10.013 M&A!

Gradient Descent, to the rescue!

Definition (Gradient Descent):

Gradient Descent (GD) is an iterative algorithm used to solve optimization problems.

- It starts with an initial set of parameters.
- It then repeatedly updates the parameters in the direction of the negative gradient of the given cost function, until it converges to a local minimum.

- The normal equation has many problems…
- But GD, on the other hand can be used to find the optimal solution even when the normal equation is not applicable.
- The main advantage of gradient descent is that it can handle very large datasets and it can also be used for non-linear models.

GD in Linear Regression

- In the case of Linear Regression, with the MSE loss defined earlier, we have

$$(a^*, b^*) = \underset{a,b}{\operatorname{argmin}} \left[L(a, b, x, y) = \frac{1}{N} \sum_{i=1}^{N} (ax_i + b - y_i)^2 \right]$$

GD in Linear Regression

We have the following derivatives with respect to a and b:

$$D_a = \frac{\partial L}{\partial a} = \frac{-2}{N} \sum_{i=1}^{N} x_i \left(y_i - (a\, x_i + b) \right)$$

$$D_b = \frac{\partial L}{\partial b} = \frac{-2}{N} \sum_{i}^{N} y_i - (a\, x_i + b)$$

GD in Linear Regression

The gradient descent **update rules** are therefore defined as

$$a \leftarrow a - \alpha D_a$$

$$a \leftarrow a + \frac{2\alpha}{N} \sum_{i=1}^{N} x_i \left(y_i - (a\, x_i + b) \right)$$

And

$$b \leftarrow b - \alpha D_b$$

$$b \leftarrow b + \frac{2\alpha}{N} \sum_{i}^{N} y_i - (a\, x_i + b)$$

With parameter α being the **learning rate** for the gradient descent, a parameter to be decided manually, later.

GD in Linear Regression

Gradient Descent Linear Regression procedure:

- We will then initialize a and b with some values (could be zero, random, or something else).
- For a given number of iterations, we will apply the two update rules defined earlier.
- (Optionally, we might decide to stop iterating, if we realize that the values of a and b are no longer changing. This is called **early stopping**, and is typically implemented by tracking the changes on each iteration and breaking if the changes are less than a threshold δ.)

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Initialize a and b as you please.

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Number of samples in dataset.

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Two possible stopping conditions, change < delta or counter > max_count

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Update using our GD update rules from earlier

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Compute change on this iteration (to decide on early stopping or not)

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Calculate new MSE value using the new parameters a and b. We also keep track of these losses for display later

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

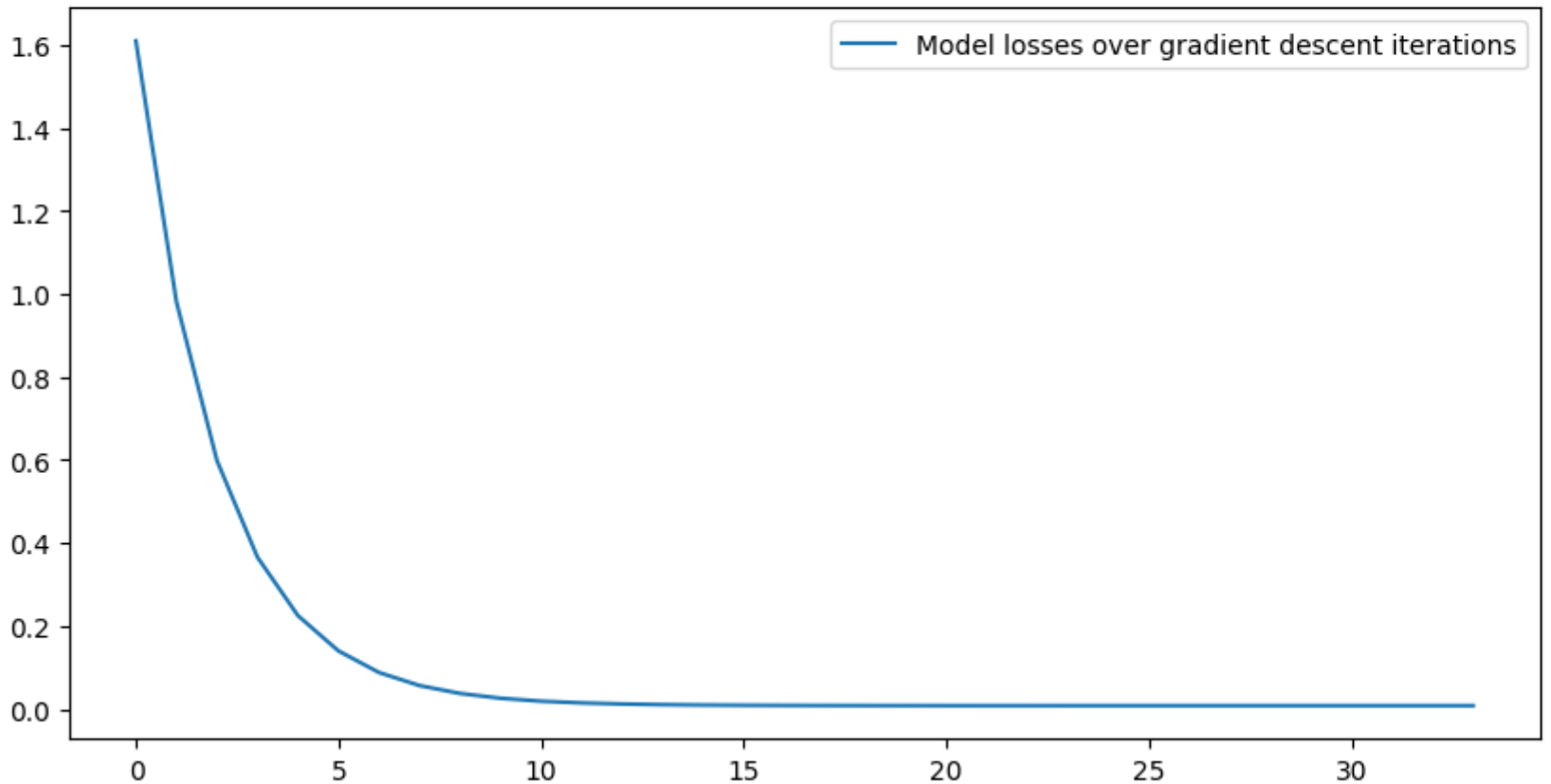
Return trained parameters and losses evolution on each iteration

```
: 1 a_gd, b_gd, losses = gradient_descent_linreg(a_0 = 0, b_0 = 0, x = inputs, y = outputs, alpha = 1e-5, delta = 1e-6)
```

```
-----
Gradients:  -342.3996226384001 3.105429920000001
New values for (a, b):  0.0034239962263840013 -3.105429920000001e-05
Change:  0.0034239962263840013
Loss:  1.61e+00
-----
Gradients:  -266.62829357559235 2.4212214483389536
New values for (a, b):  0.006090279162139925 -5.526651368338955e-05
Change:  0.0026662829357559236
Loss:  9.82e-01
-----
Gradients:  -207.62478780001163 1.8884249597020164
New values for (a, b):  0.008166527040140042 -7.415076328040972e-05
Change:  0.0020762478780001164
Loss:  5.99e-01
-----
Gradients:  -161.6784683033283 1.4735337252735503
New values for (a, b):  0.009783311723173324 -8.888610053314522e-05
Change:  0.001616784683033283
```

```python
# Display dataset
plt.figure(figsize = (10, 5))
plt.plot(losses, label = "Model losses over gradient descent iterations")

# Display
plt.legend(loc = 'best')
plt.show()
```

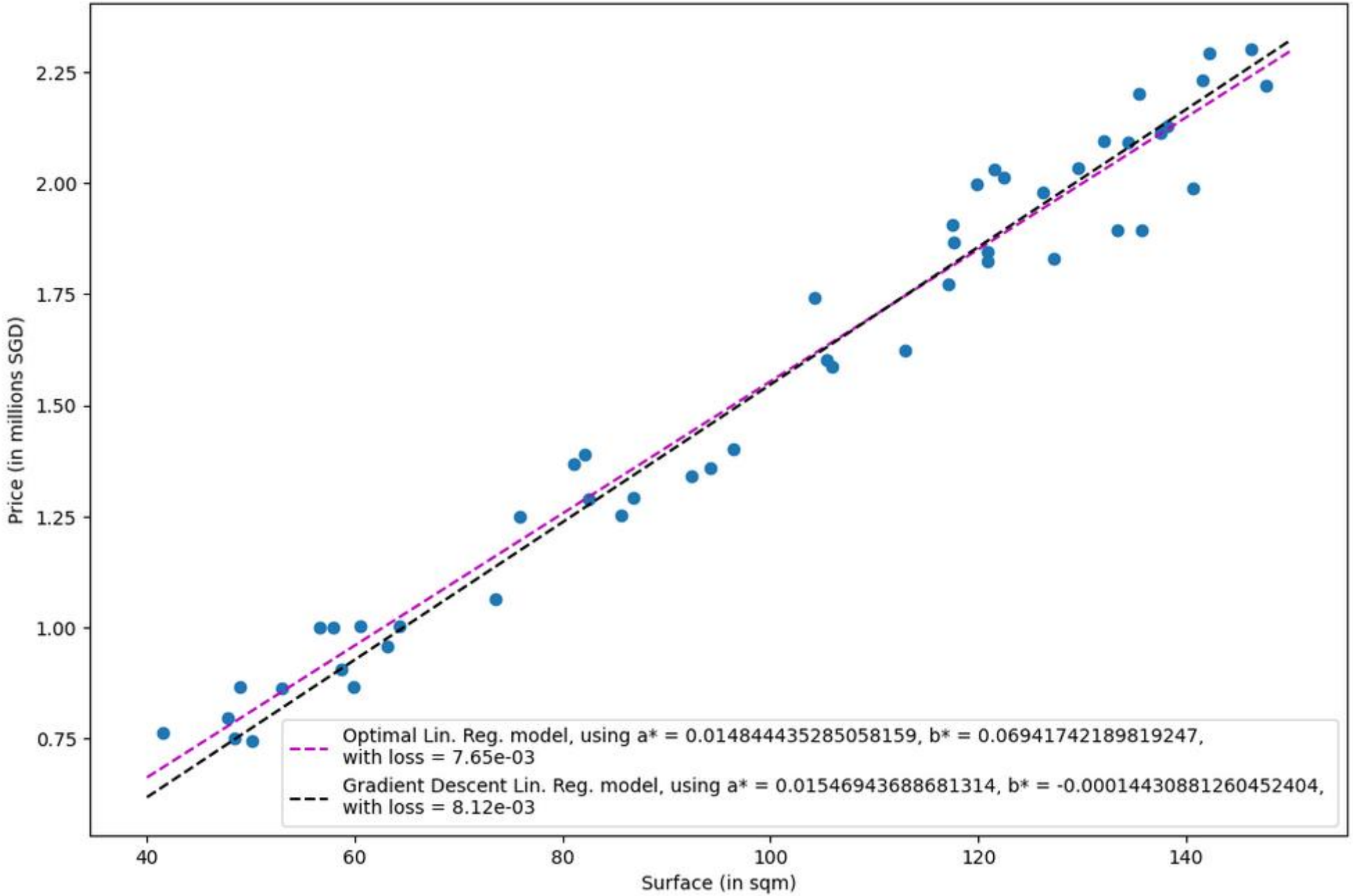


Checking for optimal parameters

- We have generated the dataset ourselves, so we know what should be the values for a and b!

```
print("Optimal a_star value: ", a_star)
print("Value for a_star, found by gradient descent: ", a_gd)
print("We used 14373/1000000 in the mock dataset generation, which is: ", 14373/1000000)
print("Optimal b_star value: ", b_star)
print("Value for b_star, found by gradient descent: ", b_gd)
print("The value we used in the mock dataset generation: ", 100000/1000000)
```

```
Optimal a_star value:  0.014844435285058159
Value for a_star, found by gradient descent:  0.015469436886813l4
We used 14373/1000000 in the mock dataset generation, which is:  0.014373
Optimal b_star value:  0.069417421898l9247
Value for b_star, found by gradient descent:  -0.0001443088126045240l
The value we used in the mock dataset generation:  0.1
```



Using Sklearn for Linear regression

- In practice, we never implement the linear regression model ourselves (but it is a good practice to try it at least once!)
- It is often faster to rely on the **sklearn** library and use the **LinearRegression** object!

```
# Creating a sklearn Linear Regression model.
# It uses the same analytical formula from earlier, i.e. W^* = (X^T X)^{-1} X^T Y.
reg = LinearRegression().fit(sk_inputs, sk_outputs)
# The coefficients for a* and b* are found using coeff_ and intercept_ respectively.
a_sk = reg.coef_[0]
b_sk = reg.intercept_
print("Optimal a_star value: ", a_star)
print("Value for a_star, found by sklearn: ", a_sk)
print("Optimal b_star value: ", b_star)
print("Value for b_star, found by sklearn: ", b_sk)
```

```
Optimal a_star value:  0.014844435285058159
Value for a_star, found by sklearn:  0.014844435285058166
Optimal b_star value:  0.06941742189819247
Value for b_star, found by sklearn:  0.06941742189818956
```

Predicting using Sklearn Linear regression

- After training, it is also good practice to check if the predictor makes sense, by asking it to predict the price of an apartment it has never seen before.
- Confirm the value manually, if possible.

```
# We can later use this Linear Regression model, to predict the price of
# a new appartment with surface 105 sqm (price in millions SGD).
new_surf = 105
pred_price = reg.predict(np.array([[new_surf]]))[0]
print(pred_price)
```

```
1.628083126829297
```

```
avg_price = 14373*new_surf + 100000
min_val = 0.9*avg_price
max_val = 1.1*avg_price
print("Min, max, avg prices: ", [min_val, max_val, avg_price])
```

```
Min, max, avg prices:  [1448248.5, 1770081.5000000002, 1609165]
```

Extension of multi-parameter Linear Regression

Let us now consider that the inputs consist of more than one parameter, e.g. each sample x_i consists of:

- The surface x_i^1,
- The distance to closest MRT x_i^2,
- Etc.

The linear regression can be simply transposed.

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

Extension of multi-parameter Linear Regression

Definition (Multi-parameter Linear Regression):

Multi-parameter Linear Regression is a model, which **assumes that there is a linear relationship between inputs and outputs.**

It therefore consists of several trainable parameters $(a_1, a_2, \dots a_K, b)$, to be freely chosen.

These will connect any input $x_i = \left(x_i^1, x_i^2, \dots, x_i^K\right)$ to its respective output y_i, with the following equation:

$$y_i \approx \sum_{k=1}^{K} a_k \, x_i^k \; + \; b$$

Extension of multi-parameter Linear Regression

The MSE loss then becomes:

$$L = \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{k=0}^{K} a_k x_i^k + b - y_i\right)^2$$

And the optimization problem related to training is then:

$$(a_0^*, a_1^*, \ldots, a_K^*, b^*) = \arg\min_{a_0, a_1, \ldots, a_K, b} L$$

Extension of multi-parameter Linear Regression

We could then try to compute the new normal equation for this problem (good practice for your optimization skills, try it out!)

Or, and it is often preferable, we could then define gradient descent update rules with respect to every trainable parameter $(\boldsymbol{a_1}, \boldsymbol{a_2}, \ldots \boldsymbol{a_K}, \boldsymbol{b})$ like before.

However, we leave this implementation for students who would like to practice. (or it might be the homework for this week!)

From Linear to Polynomial Regression

Definition (Polynomial Regression with degree K):

Polynomial Regression is a model, which **assumes that there is a polynomial relationship between inputs and outputs,** which can be defined as a **polynomial function of degree K.**

It therefore consists of several trainable parameters $(a_1, a_2, \dots a_K, b)$, to be freely chosen.

These will connect any input $x_i = \left(x_i^1, x_i^2, \dots, x_i^K\right)$ to its respective output y_i, with the following equation:

$$y_i \approx \sum_{k=1}^{K} a_k \, (x_i)^k \; + \; b$$

The hyperplane concept

Definition (Hyperplanes in Linear Algebra):

A **hyperplane** is a subspace of one dimension less than the ambient space.

In the case of linear regression with one input feature and one output feature, the ambient space is of dimension 2 (that is 1+1).

The Linear Regression model equation, is defined as

$$y = ax + b$$

This is the equation of a line, which is a 1D subspace, therefore an hyperplane of the 2D ambient space.

The same happens with the polynomial regression, but at a higher degree.

From Linear to Polynomial Regression

We will follow the same steps as before with linear regression, starting with a mock dataset.

- This will be different compared than what has been implemented in notebooks 1 and 2, as we will generate prices y_i as a polynomial function of the surfaces x_i.
- We will assume that the function $y = f(x)$, giving the price of an apartment with surface x, is defined as a polynomial function with degree 3.

$$y = f(x) = 100000 + 14373x + 3x^3$$

- In addition, we will add a random noise to the final pricing, with a +/- 10% drift as before.

From Linear to Polynomial Regression

We will follow the same steps as before with linear regression, starting with a mock dataset.

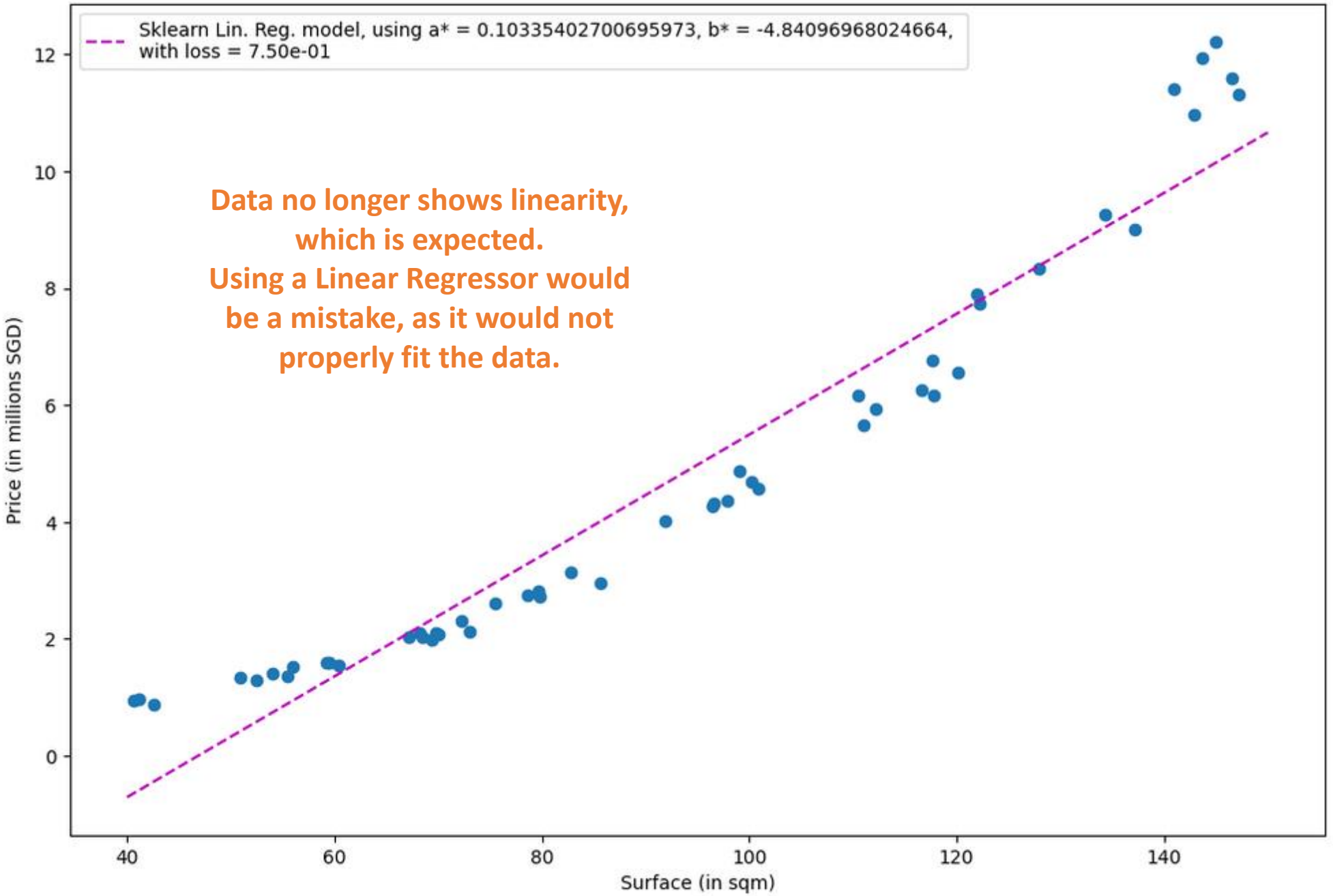
- This will be different compared than what has been implemented in notebooks 1 and 2, as we will generate prices y_i as a polynomial function of the surfaces x_i.
- We will assume that the function $y = f(x)$, giving the price of an apartment with surface x, is defined as a polynomial function with degree 3.

$$y = f(x) = 100000 + 14373x + 3x^3$$

- In addition, we will add a random noise to the final pricing, with a +/- 10% drift as before.

From Linear to Polynomial Regression

```python
# All helper functions
def surface(min_surf, max_surf):
    return round(np.random.uniform(min_surf, max_surf), 2)
def price(surface):
    # Note: this has changed and is now a polynomial function.
    return round((100000 + 14373*surface + 3*surface**3)*(1 + np.random.uniform(-0.1, 0.1)))/1000000
def generate_datasets(n_points, min_surf, max_surf):
    x = np.array([surface(min_surf, max_surf) for _ in range(n_points)])
    y = np.array([price(i) for i in x])
    return x, y
def linreg_matplotlib(a, b, min_surf, max_surf, n_points = 50):
    x = np.linspace(min_surf, max_surf, n_points)
    y = a*x + b
    return x, y
def loss_mse(a, b, x, y):
    val =  np.sum((y - (a*x + b))**2)/x.shape[0]
    return '{:.2e}'.format(val)
```



Polynomial Regression with sklearn

Sklearn treats polynomial regression as a **multi-parameter linear regression**, with **polynomial features**.

Polynomial features: reworking the inputs so that

$$\tilde{X} = \begin{pmatrix} x_1 & \cdots & (x_1)^K \\ \vdots & \ddots & \vdots \\ x_N & \cdots & (x_N)^K \end{pmatrix}$$

And $\tilde{x}_{i,j} = (x_i)^j$

```
1 # Preparing polynomial features for our dataset
2 n_degree = 3
3 sk_poly = PolynomialFeatures(degree = n_degree, include_bias = False)
4 sk_poly_inputs = sk_poly.fit_transform(sk_inputs.reshape(-1, 1))
5 print(sk_poly_inputs)
```

```
[[5.24800000e+01 2.75415040e+03 1.44537813e+05]
 [1.47190000e+02 2.16648961e+04 3.18885606e+06]
 [1.20160000e+02 1.44384256e+04 1.73492122e+06]
 [7.86600000e+01 6.18739560e+03 4.86700538e+05]
 [1.17840000e+02 1.38862656e+04 1.63635754e+06]
 [1.27960000e+02 1.63737616e+04 2.09518653e+06]
 [1.11010000e+02 1.23232201e+04 1.36800066e+06]
 [8.56100000e+01 7.32907210e+03 6.27441862e+05]
 [1.17660000e+02 1.38438756e+04 1.62887040e+06]
 [6.71300000e+01 4.50643690e+03 3.02517109e+05]
 [6.81600000e+01 4.64578560e+03 3.16656746e+05]
 [4.26400000e+01 1.81816960e+03 7.75267517e+04]
 [5.08600000e+01 2.58673960e+03 1.31561576e+05]
 [7.30500000e+01 5.33630250e+03 3.89816898e+05]
 [1.10490000e+02 1.22080401e+04 1.34886635e+06]
 [7.54400000e+01 5.69119360e+03 4.29343645e+05]
 [6.04000000e+01 3.64816000e+03 2.20348864e+05]
```

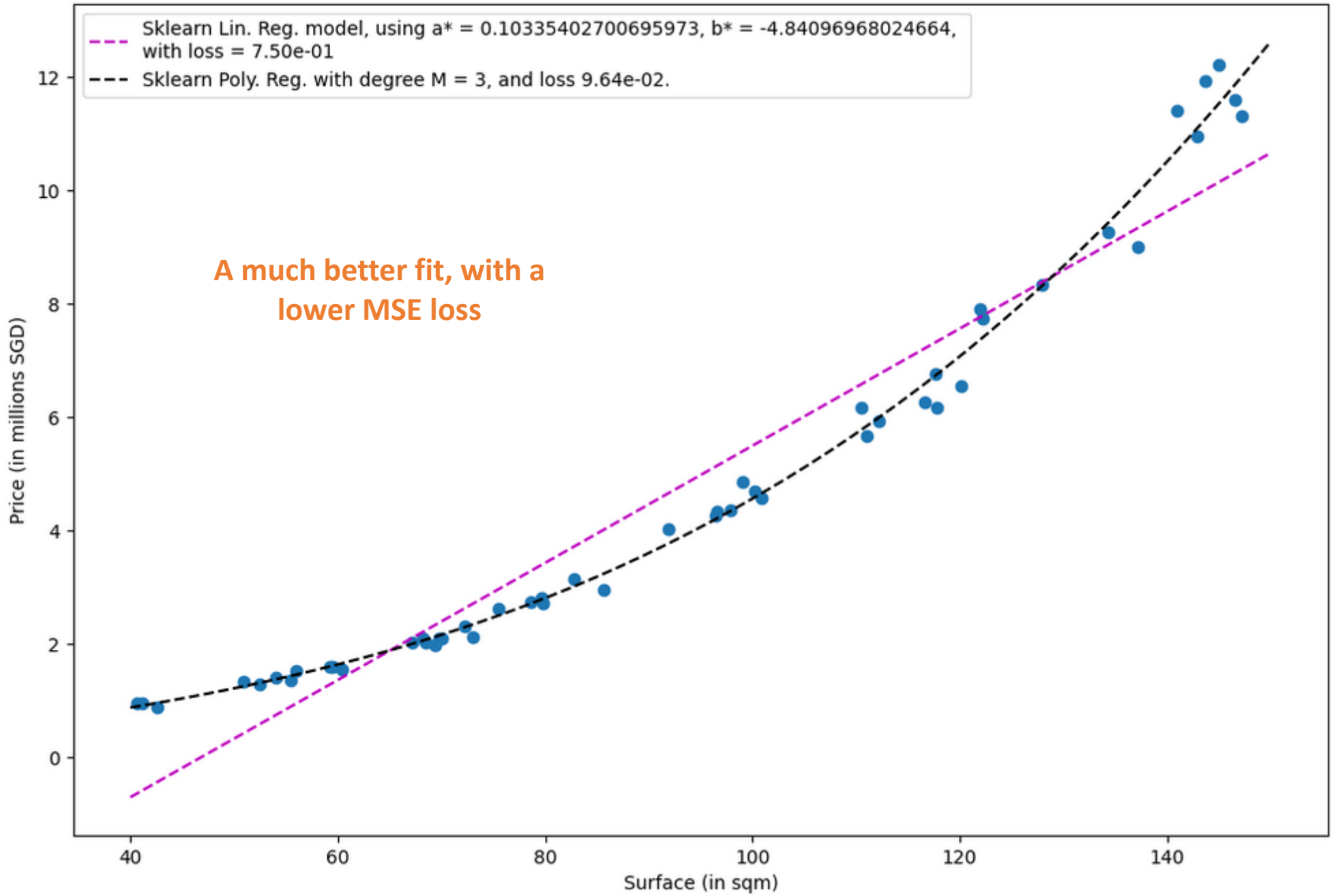
Polynomial Regression with sklearn

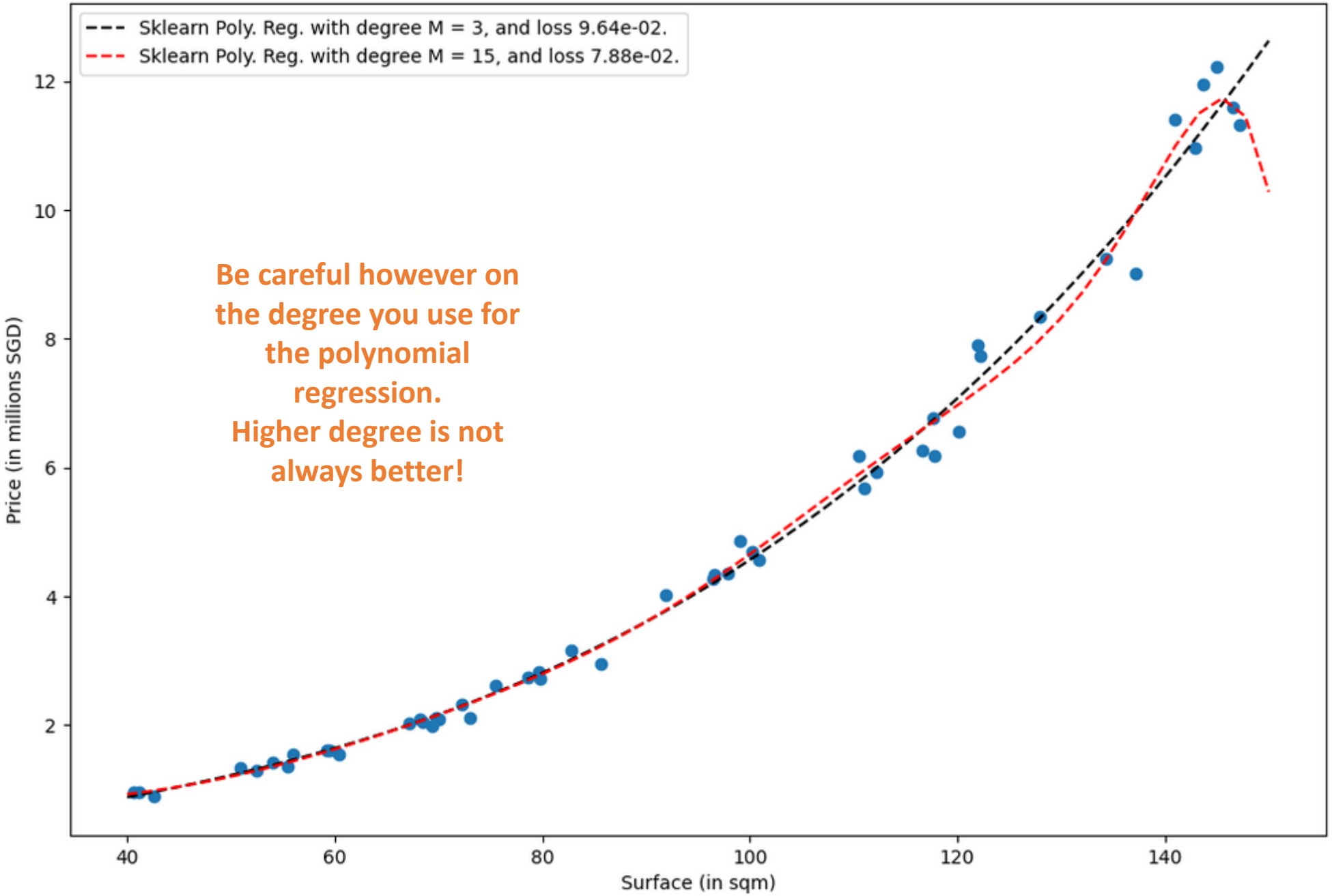
We then proceed normally, assuming $\tilde{X}$ are our new inputs.

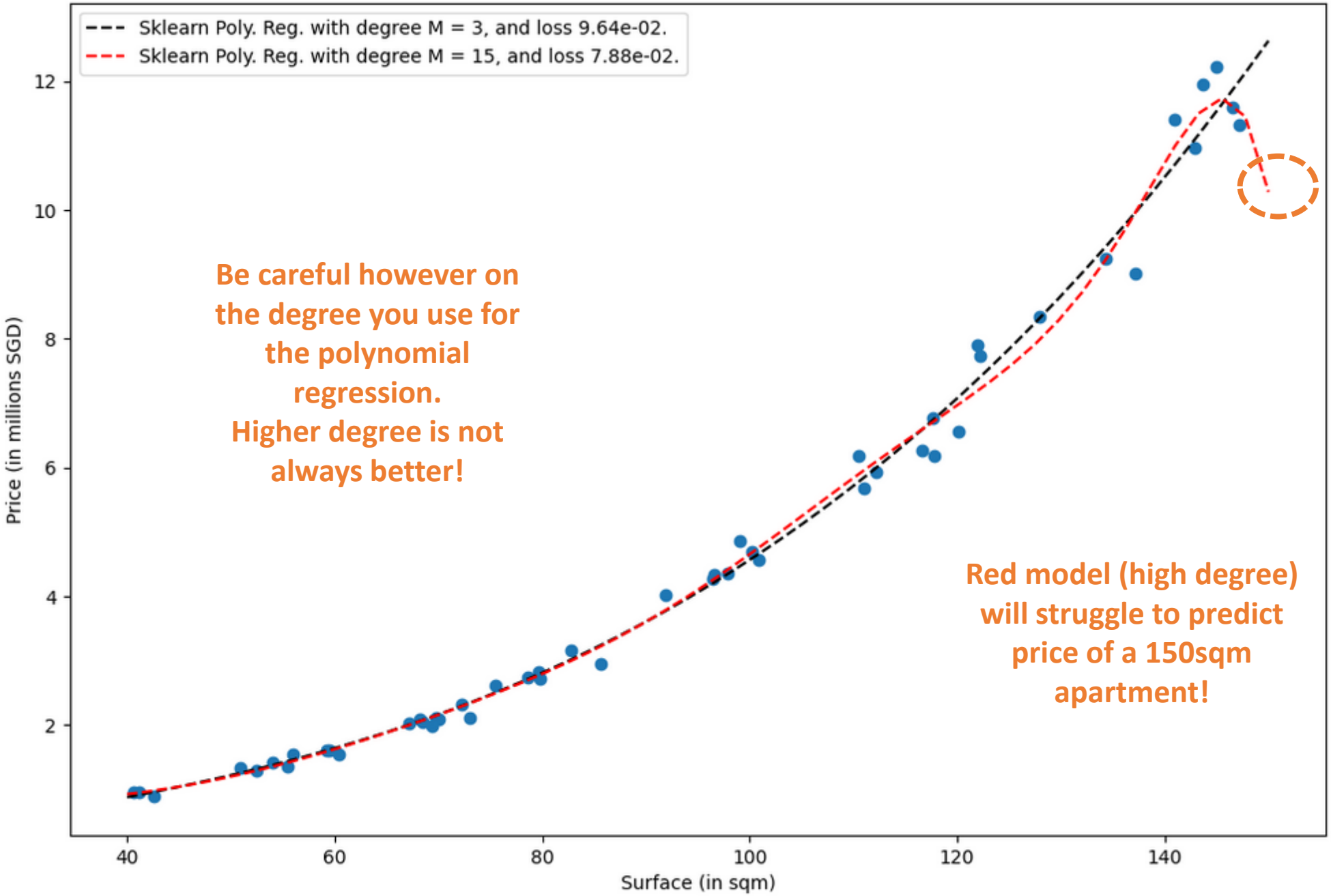
Sklearn then automatically adjusts and produces the right number of parameters for the model.

```
1  # Training a Polynomial Regressor
2  poly_reg_model = LinearRegression()
3  poly_reg_model.fit(sk_poly_inputs, sk_outputs)
4  a_sk_poly = poly_reg_model.coef_
5  b_sk_poly = poly_reg_model.intercept_
6  print(a_sk_poly, b_sk_poly)
```

```
[ 2.38010878e-02 -1.30213791e-04  3.58102988e-06] -0.09785310239196843
```



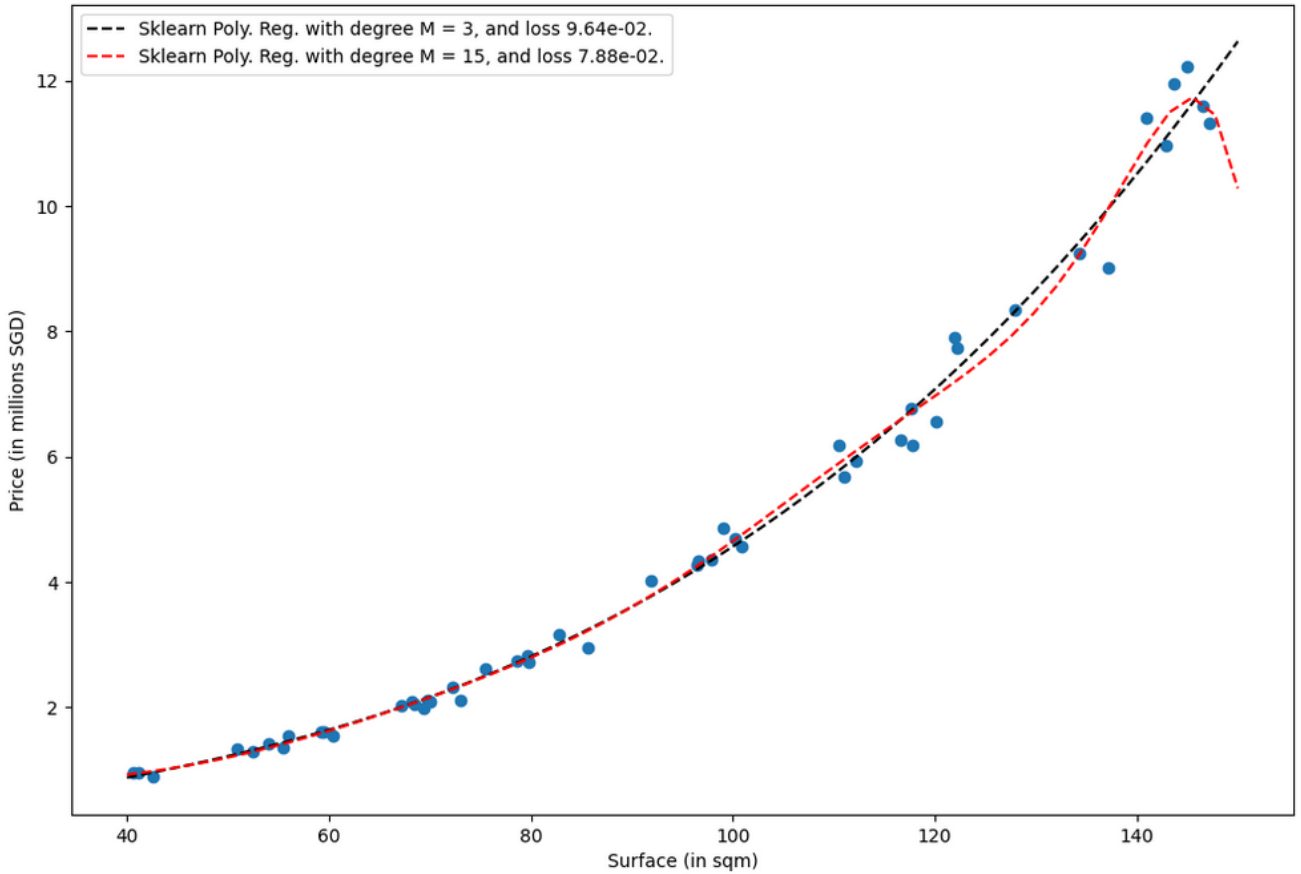




Overfitting vs. degree of polynomial

- Red curve shows what happens when using a degree K that is too high compared to the relationship connecting inputs and outputs (it was 3).

- This phenomenon is called **overfitting**.

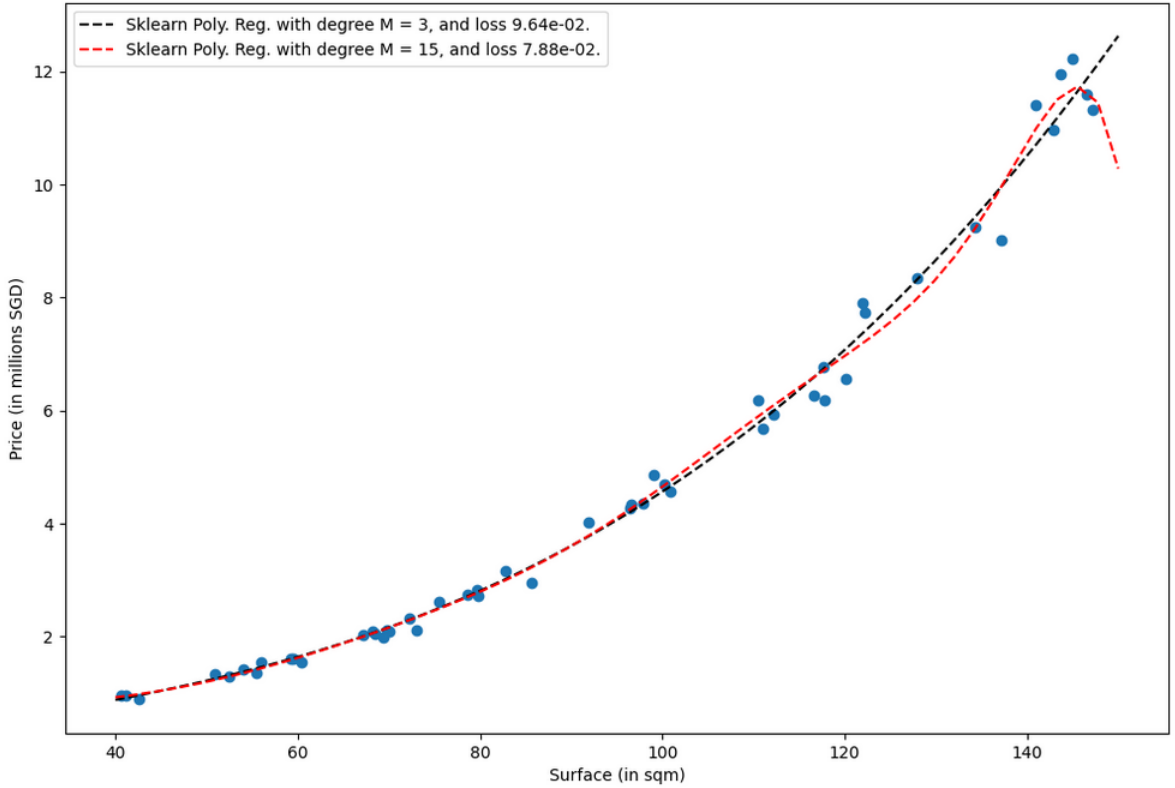


Overfitting vs. degree of polynomial

Definition (Overfitting):

Overfitting is a phenomenon that occurs in machine learning when a model is trained too well on the training data.

As a result, it performs poorly on new, unseen data (here an apartment with 150sqm surface, which we had not in the training dataset).



Overfitting vs. degree of polynomial

Definition (Overfitting):

Overfitting is a phenomenon that occurs in machine learning when a model is trained too well on the training data.

As a result, it performs poorly on new, unseen data (here an apartment with 150sqm surface, which we had not in the training dataset).

This typically happens when

- a model is trained with too many features
- or too much data,

It becomes too complex for the task at hand, resulting in poor **generalization** to new data.

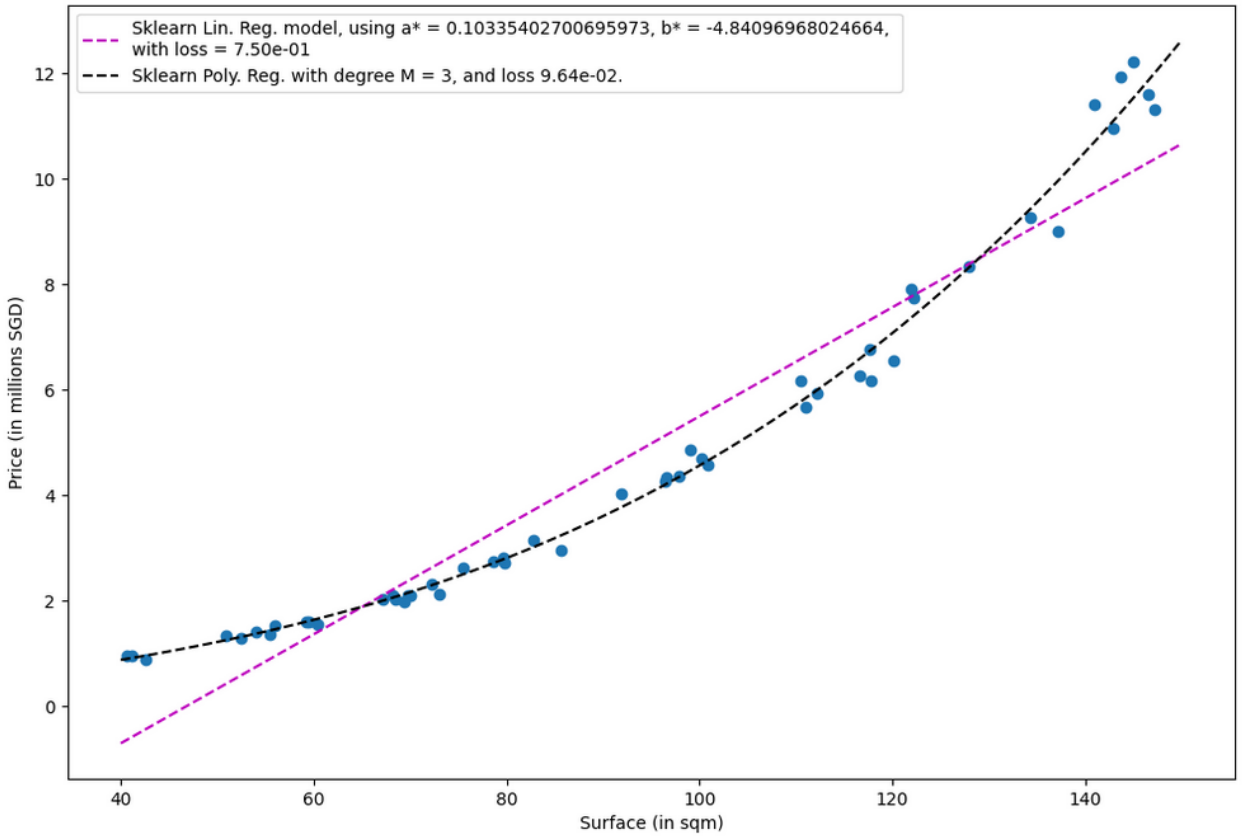
In other words, it memorizes the noise in the training data rather than learning the correct underlying pattern.

Underfitting vs. degree of polynomial

Definition (Underfitting):

Underfitting is a phenomenon that occurs in machine learning when a model is not capable enough to capture the underlying pattern of the data.

For instance, here, Linear Regression (which is Polynomial Regression with $K = 1$) could not fit the data well.



Underfitting vs. degree of polynomial

Definition (Underfitting):

Underfitting is a phenomenon that occurs in machine learning when a model is not capable enough to capture the underlying pattern of the data.

For instance, here, Linear Regression (which is Polynomial Regression with $K = 1$) could not fit the data well.

This can happen when a model

- is trained with too few features
- or too little data,
- or when the model is not powerful enough to capture the complexity of the task.

Generalization

Definition (Generalization):

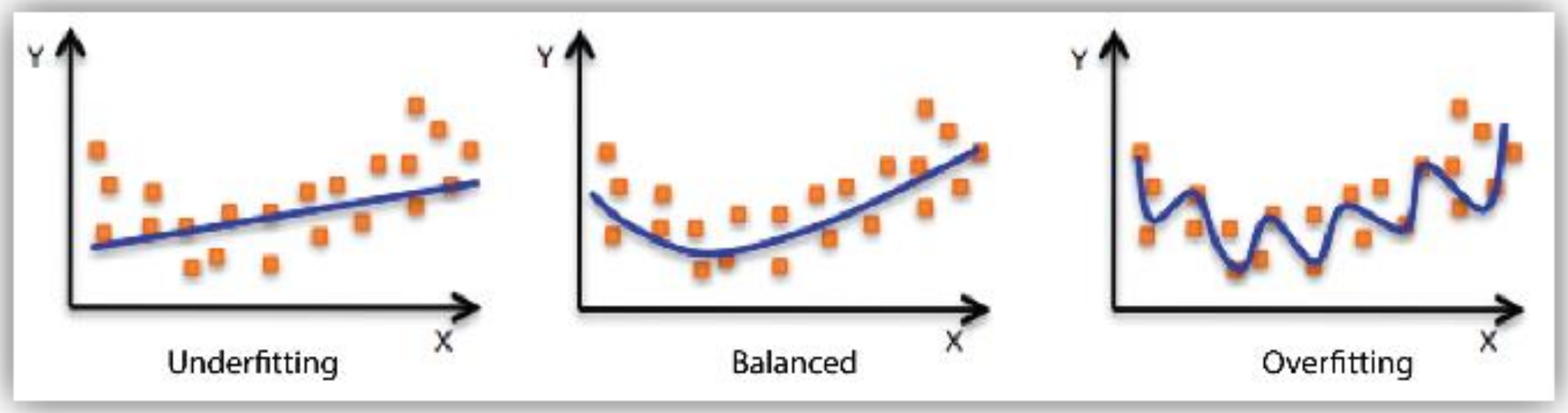
Generalization is considered the "holy grail" of machine learning because it is the ultimate goal of any machine learning model.

Even though the model was trained on seen data we collected, the purpose of a machine learning model is to make accurate predictions on new, unseen data.

If a model can make accurate predictions on data it has not seen before, it is said to have **generalized well**, and can then be used in the real world for solving the problem it was designed to solve.

Generalization vs. Overfitting/Underfitting

- In general, having a model that is **underfitting** or **overfitting** will lead to poor **generalization**.
- We would very much prefer to fit the data “just right”.



While we are at it…

In general, we want to:

- Train the model was trained on data we collected,
- Confirm that the model can generalize and make accurate predictions on new, unseen data, after training.

Problem: We would prefer to test the model before releasing it in the wild (what if it is wrong?)

Problem #2: Testing its generalization capabilities on the same data that was used for training would NOT confirm generalization.

Solution: Use some of our seen data to train the model, and some of the data (that has not been used for training and is therefore unseen to the model yet) for evaluating its generalization.

Train and test split

Definition (Train and Test Split):

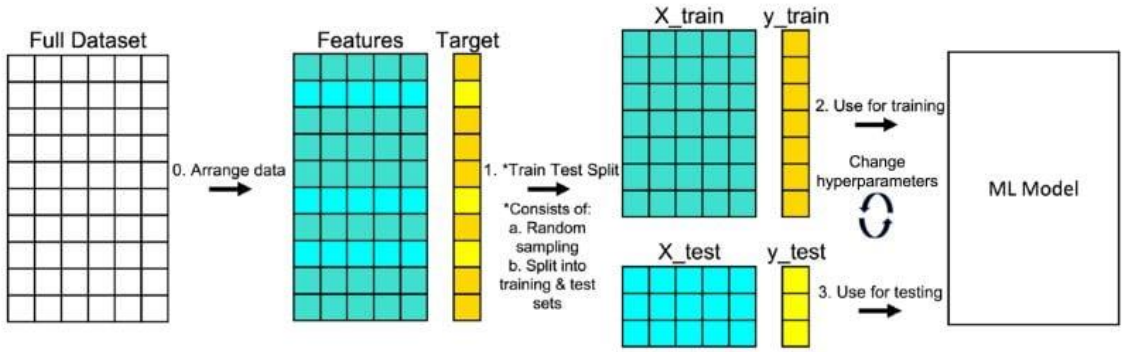
Train and test split is a method used in machine learning to divide a dataset into two subsets:

- The **training set** is used to train the model
- The **test set** is used to evaluate the performance of the trained model.

The idea is to use a portion of the dataset for training and a separate portion for testing so that the model can be evaluated on unseen data.

The typical split is to use 80% of the data for training and 20% for testing.

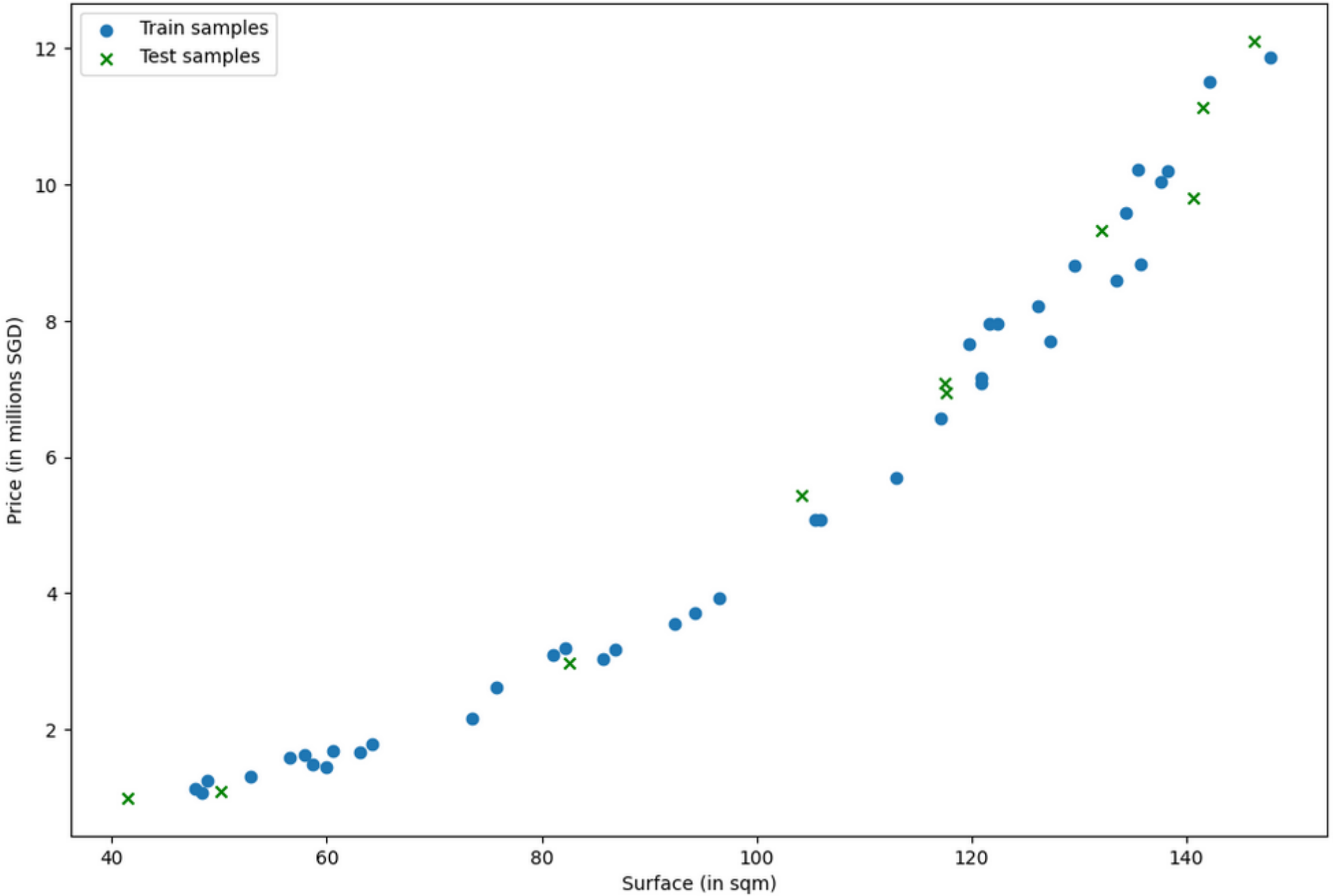
This can however vary depending on the specific use case and the size of the dataset.



Train and test split

```
# 80% of the samples will be used for training,
# and the remaining 20% will be used to evaluate generalization/overfitting.
ratio_train = 0.8
split_index = int(n_points*ratio_train)
# Training inputs and outputs
train_inputs = inputs[:split_index]
train_outputs = outputs[:split_index]
# Testing inputs and outputs
test_inputs = inputs[split_index:]
test_outputs = outputs[split_index:]
# Display
print(train_inputs)
print(train_outputs)
print(test_inputs)
print(test_outputs)
```

```
[ 86.83 129.6  120.89 135.48  82.17 147.74 138.25  63.07 121.6  112.95
 137.55 134.38 122.42 135.72  60.54  75.81  81.02 127.31  56.62  58.69
  48.93  73.57 126.16  57.92  47.77 117.12  59.91 105.88  85.68  96.49
  64.27 119.81 133.44 142.18 120.95  92.42  94.22 105.4   48.36  52.92]
[ 3.171633  8.805667  7.166722 10.224944  3.195151 11.87205  10.206911
  1.676214  7.954078  5.696169 10.049157  9.588631  7.968783  8.827882
  1.693317  2.6239    3.094831  7.700582  1.597057  1.489182  1.247223
  2.166055  8.212138  1.627403  1.12722   6.560241  1.450404  5.072051
  3.031968  3.938955  1.782147  7.652644  8.58658  11.518957  7.088832
  3.563134  3.702923  5.087156  1.071143  1.308982]
[146.31 104.17  50.17  41.5  132.06 140.63 117.5   82.57 117.63 141.56]
[12.111945  5.438864  1.088438  0.998857  9.333922  9.801835  7.095279
  2.982004  6.953392 11.129658]
```

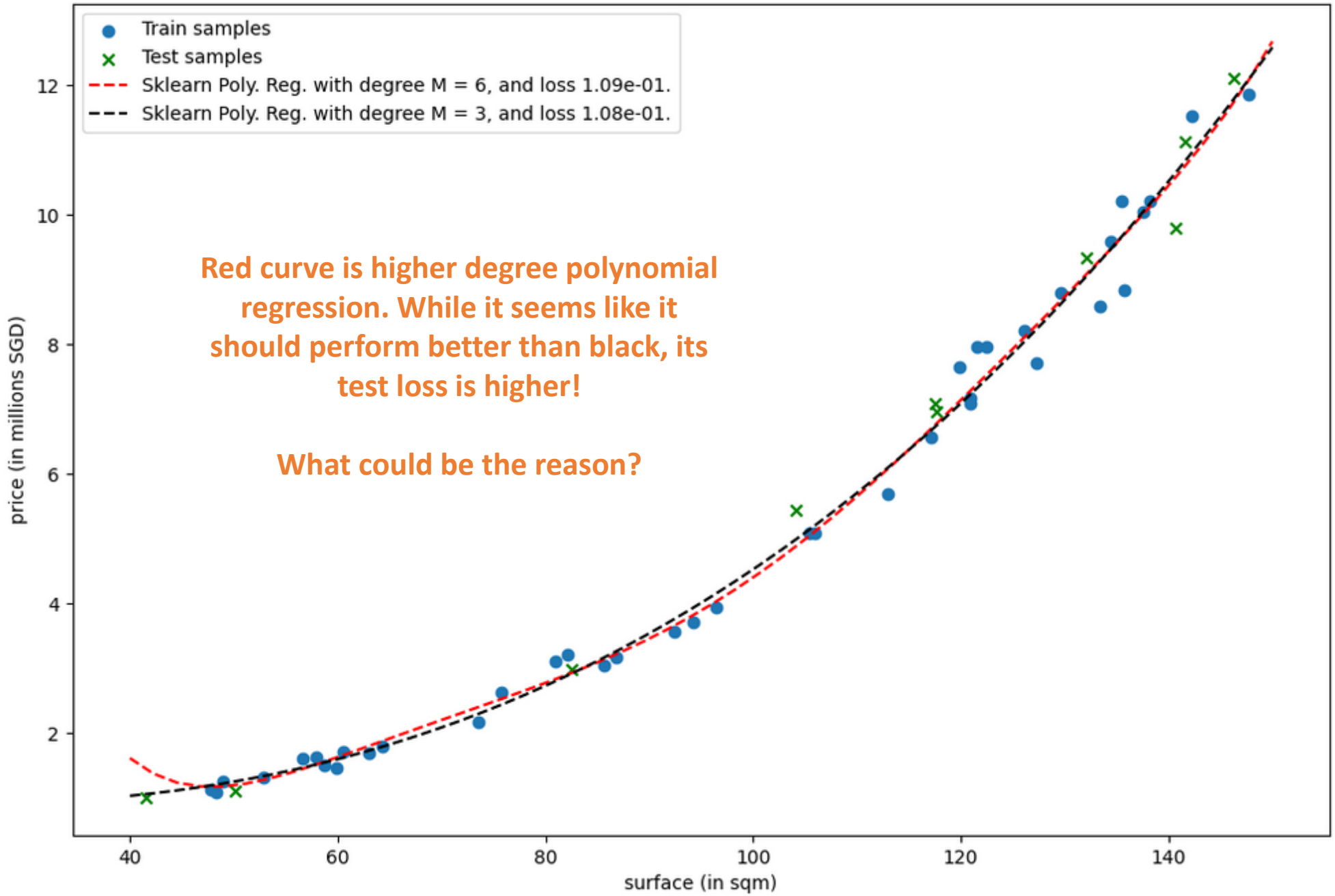


Training the Polynomial Regressor, again

- This time, we only train the polynomial regressor on the training data.
- After training, evaluate loss using the test data.

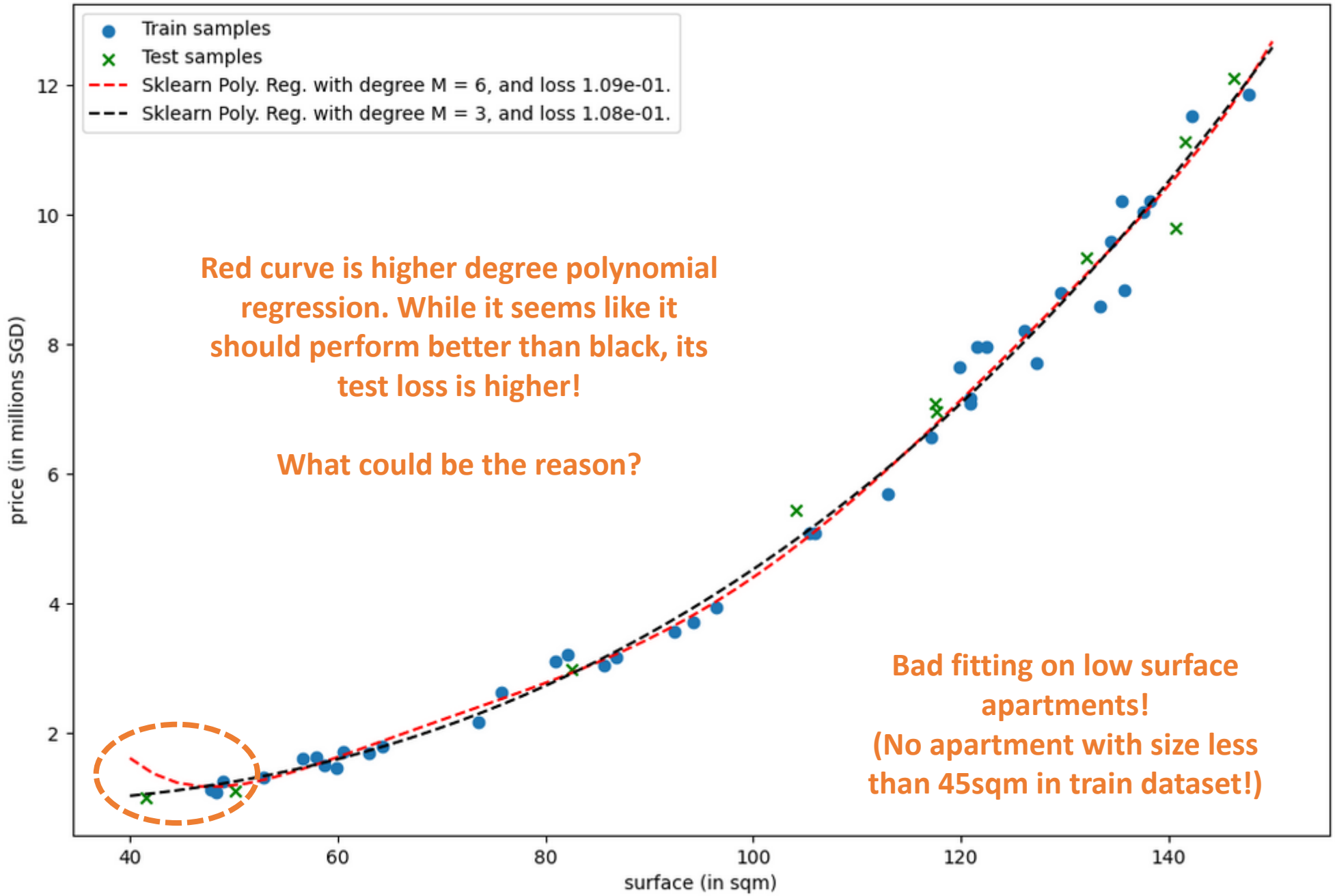
```
# Training a Polynomial Regressor
n_degree = 6
sk_inputs = np.array(train_inputs).reshape(-1, 1)
sk_outputs = np.array(train_outputs)
sk_poly = PolynomialFeatures(degree = n_degree, include_bias = False)
sk_poly_inputs = sk_poly.fit_transform(sk_inputs.reshape(-1, 1))
poly_reg_model = LinearRegression()
poly_reg_model.fit(sk_poly_inputs, sk_outputs)
a_sk = poly_reg_model.coef_
b_sk = poly_reg_model.intercept_
print(a_sk, b_sk)
```

```
[-5.31103644e+00  1.55941662e-01 -2.34706297e-03  1.92370550e-05
 -8.12548986e-08  1.38734052e-10] 73.25495081585393
```



Red curve is higher degree polynomial regression. While it seems like it should perform better than black, its test loss is higher!

What could be the reason?



Generalization vs. Train/Test Distributions

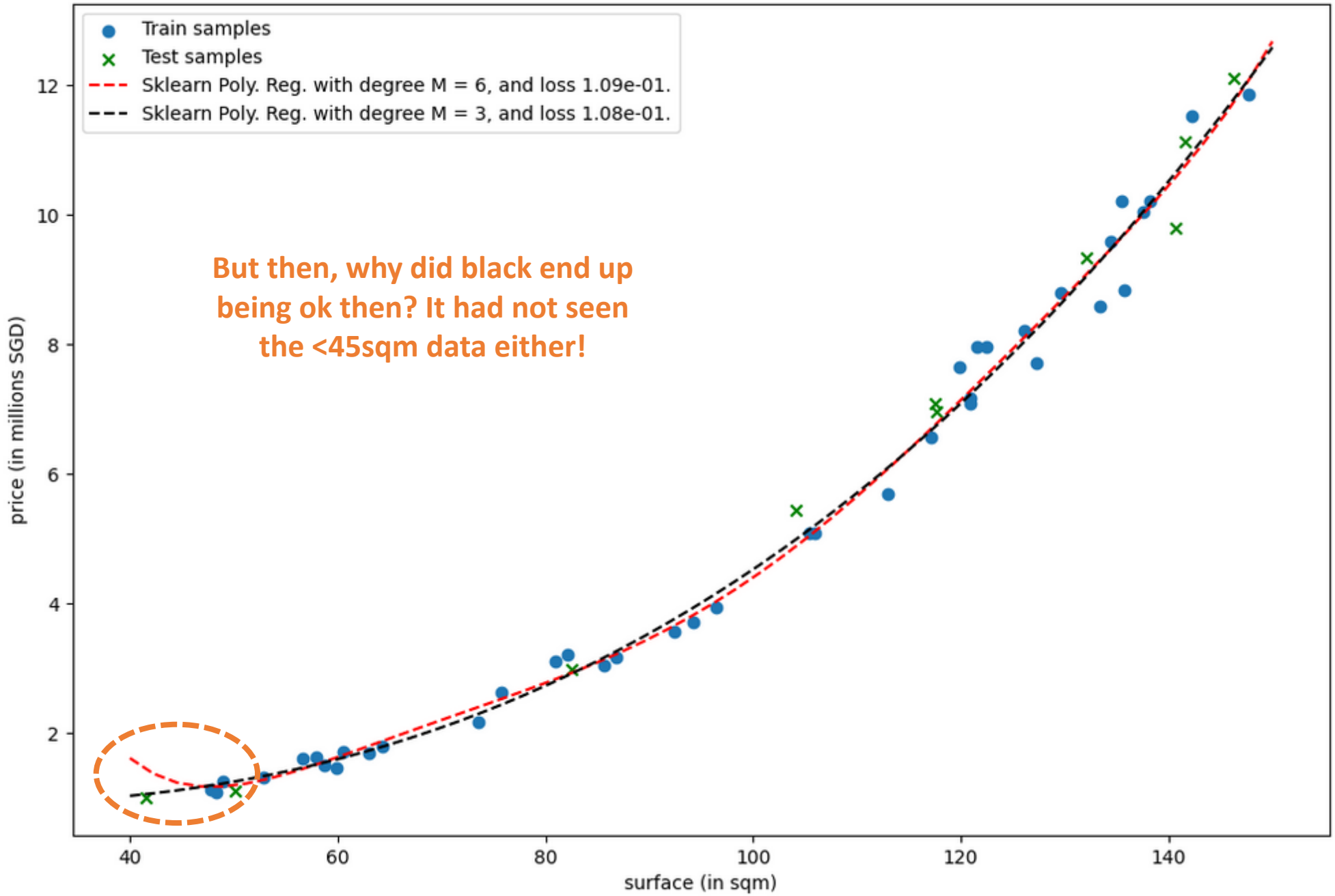
Definition (The need for train and test distribution similarity):

In order to generalize well in is important that the **train and test set are following similar distributions**.

$$P_{train} \approx P_{test}$$

Here, we ended up having a problem, as our training dataset did not contain any apartments with size lower than 45sqm.

I mean, you would not train a model on Singaporean apartments and use it to predict the price of apartments in Kuala Lumpur, right?



Regularization

Definition (Regularization):

Regularization consists of helping the model avoid overfitting (which is usually more prominent than underfitting).

It can typically be done by making sure the model is not too complex for the data, but many other approaches exist.

In general,

- the more complex the data is (in terms of features), the more complex the model can be.
- the more data you have (in terms of quantity of individual samples in dataset), the more complex the model can be.

Regularization

Definition (Regularization):

Regularization consists of helping the model avoid overfitting (which is usually more prominent than underfitting).

It can typically be done by making sure the model is not too complex for the data, but many other approaches exist.

A more common approach consists of **adding a penalty term to the loss function**.

The penalty term, also known as a **regularization term**, discourages the model from assigning too much weight to any one feature, i.e. making a certain parameter a_k too prominent (leading to overfitting).

Regularization

Definition (Ridge Regression):

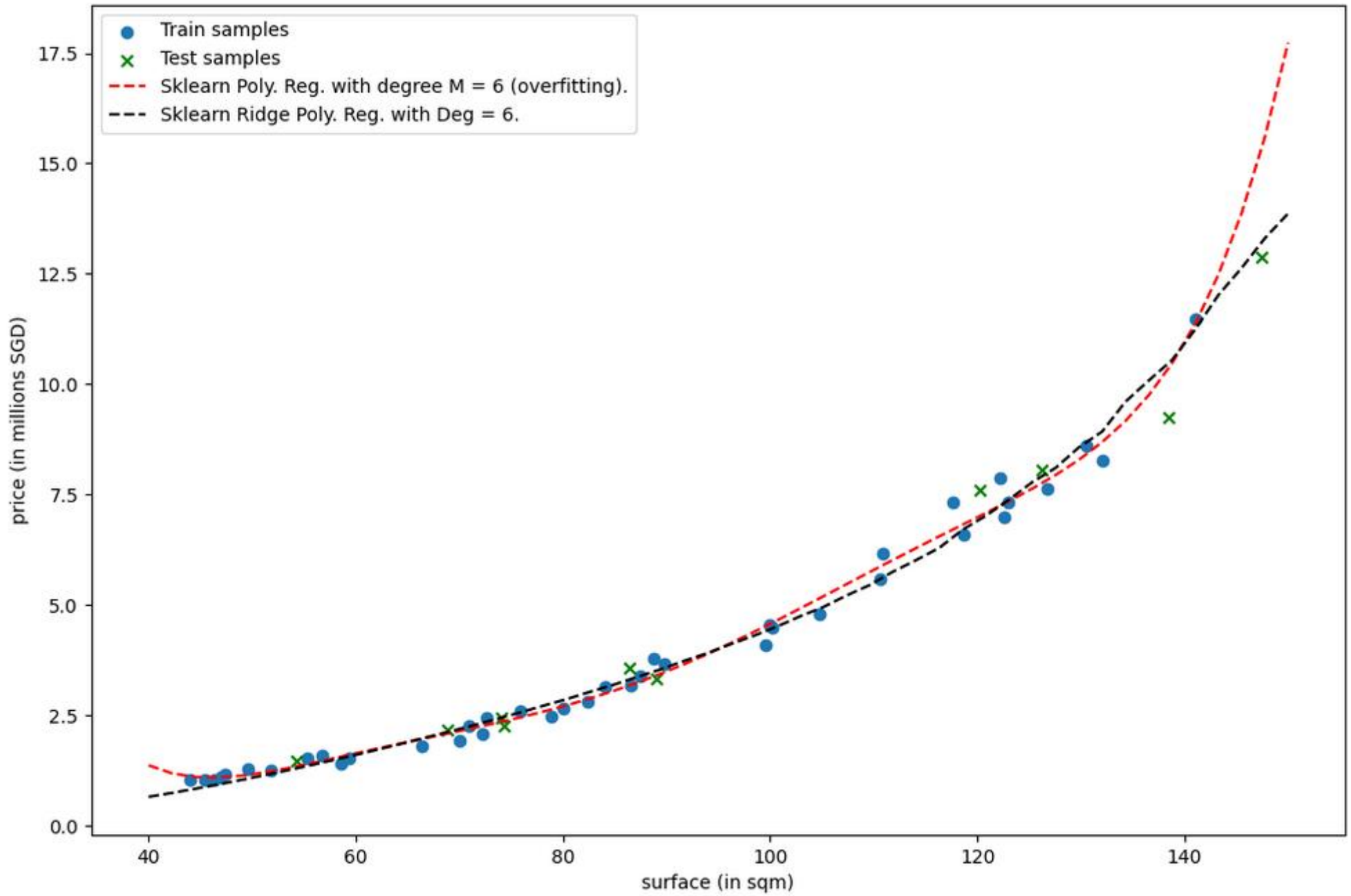
The **Ridge Polynomial Regressor is a polynomial regressor, like described earlier.**

Its only difference is **the loss function includes an additional regularization term $\lambda\, R(a, b)$.**

This regularization term consists of the **sum of the squared values for the trainable parameters**, i.e.:

$$\lambda\, R(a,b) = \frac{\lambda}{2N}\left(\sum_{k}^{M} (a_k)^2 \; + \; (b)^2\right)$$

- This regularization term encourages the model to assign low values to the coefficients of the model.
- This will, in turn, lead to less overfitting. Indeed, overfitting often occurs when high values are assigned to coefficients for high degrees (i.e. coefficients a_k with for high k values).



Regularization

Definition (Ridge Regression):

The **Ridge Polynomial Regressor is a polynomial regressor, like described earlier.**

Its only difference is **the loss function includes an additional regularization term $\lambda\, R(a, b)$.**

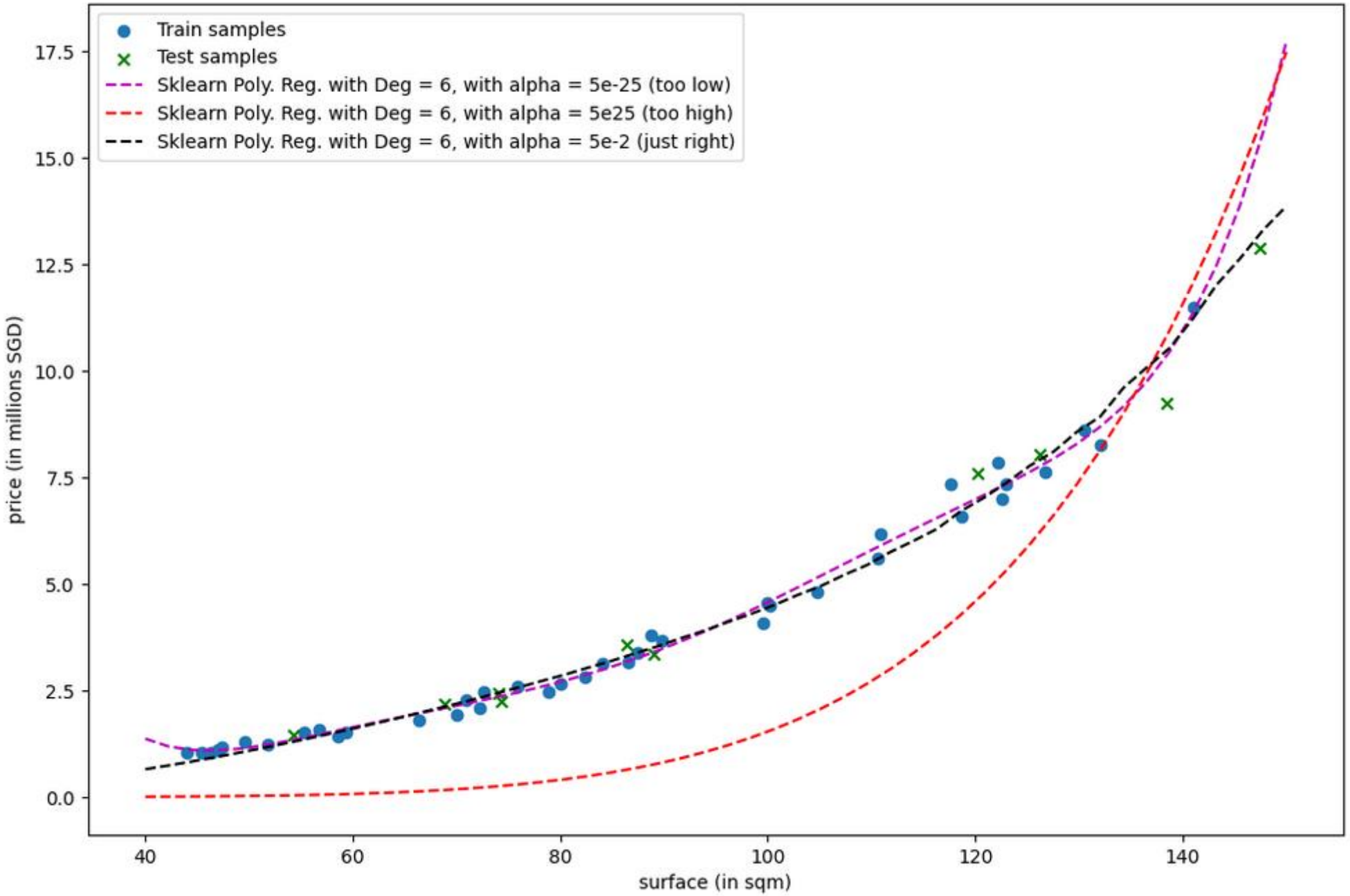
This regularization term consists of the **sum of the squared values for the trainable parameters**, i.e.:

$$\lambda\, R(a,b) = \frac{\lambda}{2N}\left(\sum_{k}^{M} (a_k)^2 \; + \; (b)^2\right)$$

Note: Just like the learning rate in GD, the parameter λ is manually decided and is used to weight the importance of the regularization term compared to the MSE.

If the **value for λ is small**, the loss function will be **mostly MSE** and our model will therefore suffer from the same **overfitting** problems as earlier.

If the **value of λ is too high** however, the model will **mostly ignore the MSE** part of the loss function, which will lead to a model not fitting (or **underfitting**) the data.



Regularization

Definition (Ridge Regression):

The **Ridge Polynomial Regressor is a polynomial regressor, like described earlier.**

Its only difference is **the loss function includes an additional regularization term $\lambda\, R(a, b)$.**

This regularization term consists of the **sum of the squared values for the trainable parameters**, i.e.:

$$\lambda\, R(a,b) = \frac{\lambda}{2N}\left(\sum_{k}^{M} (a_k)^2 \; + \; (b)^2\right)$$

Note #2: it is also possible to use the sum of **absolute** values, making a **Lasso Regressor** instead.

Why use L1 regularization over L2 regularization?

- L1 regularization results in some weights being exactly equal to zero, which can be used for feature selection.
- L2 regularization will not produce sparse models, and it will only shrink the weights.

Regularization

Definition (Ridge Regression):

The **Ridge Polynomial Regressor is a polynomial regressor, like described earlier.**

Its only difference is **the loss function includes an additional regularization term $\lambda\, R(a, b)$.**

This regularization term consists of the **sum of the squared values for the trainable parameters**, i.e.:

$$\lambda\, R(a,b) = \frac{\lambda}{2N}\left(\sum_{k}^{M} (a_k)^2 \; + \; (b)^2\right)$$

Note #3: it is also possible to use the sum of **absolute** values **AND squared** values as regularization

Essentially getting the best of both worlds, and making an **Elastic Regressor** instead!

Check it out!