Week 10: Model Selection

Polynomial regression

In week 7, we saw how to build a linear regression model. Recall that these types of regression models find a **linear relationship** between the features and label. For example, if we have two features then the linear regression equation would be:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

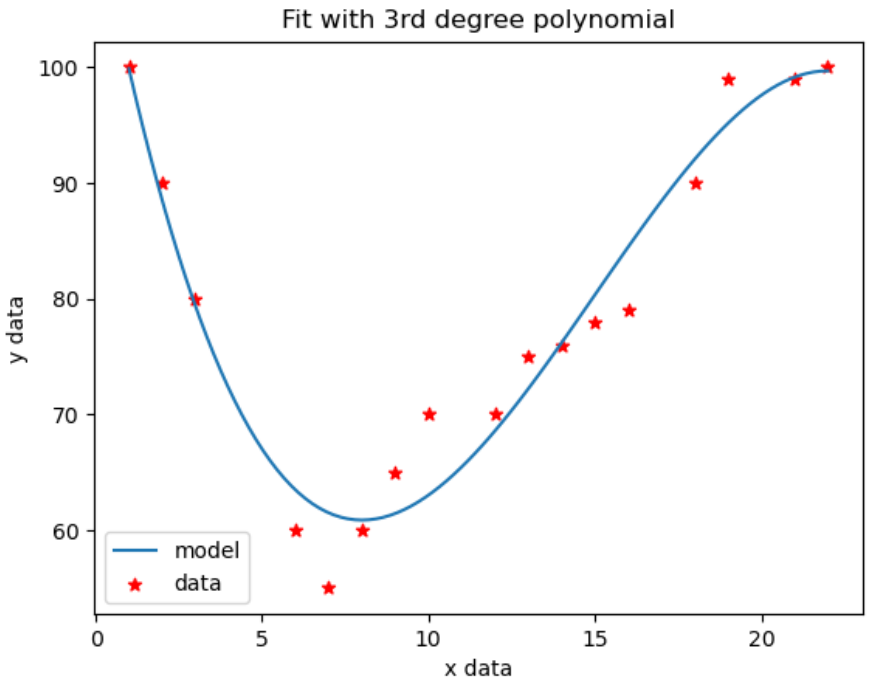
However, often you have data that includes **nonlinearities** and sticking with a linear regression model can result in these relationships being missed and your model performing lower than what it could if a non-linear relationship was modelled.

Polynomial Regression:

Polynomial regression models a non-linear relationship between our features and the label. The most general equation for polynomial regression is:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + ... + \beta_p x^p + \epsilon$$

where p is the degree of the polynomial we are using to model our data.



In the example above we have modelled the relationship using a **3rd degree polynomial** i.e. we have used the equation:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \epsilon$$

The model has been trained and an optimal $\beta_0, \beta_1, \beta_2, \beta_3$ has been found that allows the model to make the most optimal predictions of the target variable given a set of 3 features.

This lesson you will be carrying out polynomial regression on the *BatonRouge.csv* dataset by considering not only the *SQFT* but the $SQFT^2$ when it comes to estimating *Price*

Note: non-linearities can also be included in your model equation by multiplying features together i.e. $x_1 x_2$ or $x_1^2 x_4$ - these can be used to model more complex relationships

Relation to Linear Regression

The polynomial regression equation bares many similarities to the linear regression equation. This is not a fluke!

Polynomial regression **is a special case of linear regression**. This becomes much easier to see if you rewrite things in matrix form.

The formula:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + ... + \beta_p x^p + \epsilon$$

can be rewritten as:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & ... & x_1^p \\ 1 & x_2 & ... & x_2^p \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & ... & x_n^p \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

where n is the number of training samples you have.

We can write this as:

$$\mathbf{y} = \mathbf{X}\beta + \epsilon$$

This is exactly what linear regression is! Polynomial regression is linear regression where you have features being fitted as polynomials.

What this means is that we can find the optimal β values using OLS just like you did with linear regression.

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

☆☆ SQFT squared feature

You are going to be building a new linear regression model that has SQFT^2 has an additional feature. The model has the equation:

$$\text{Price} = \beta_0 + \beta_1 \text{SQFT} + \beta_2 \text{SQFT}^2$$

Your task here is to:

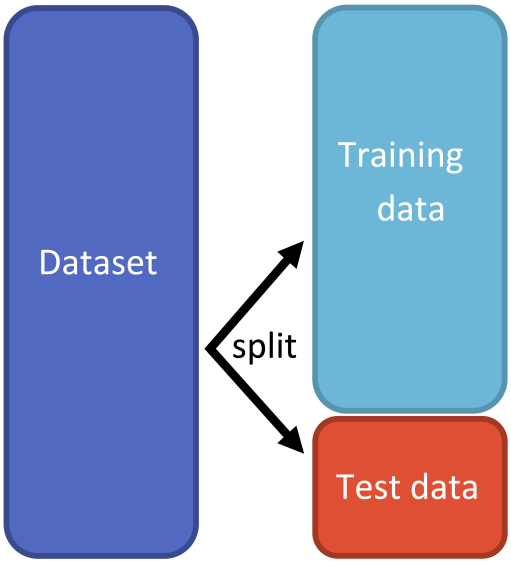
1. Create a new column called `SQFT^2` inside the *BatonRouge* DataFrame that is equal to the SQFT squared
2. Calculate the correlation between `Price` and all the other variables using `corr()`
3. Print out the results in **descending** order

Your output should look like this:

```
Price         X.XXXXXX
SQFT^2        X.XXXXXX
SQFT          X.XXXXXX
Baths         X.XXXXXX
Bedrooms      X.XXXXXX
Fireplace     X.XXXXXX
Waterfront    X.XXXXXX
Pool          X.XXXXXX
Style         X.XXXXXX
DOM           X.XXXXXX
Occupancy    -X.XXXXXX
Age          -X.XXXXXX
Name: Price, dtype: float64
```

Training and test data

When building models, it's important to have a way of evaluating them so you know whether they are performing well or not. To do this, we break the data up into a **training set** and a **test set**.



A **training dataset** is the portion of all available data that we used to train a model when we perform `.fit()`. This is what your model learns from.

You use **test data** to evaluate how well your model performs. You perform `.predict()` to get the model output for test dataset, and you compare these results with the true y values.

Analogy

During semester you learn on lots of practice questions. Each week you (hopefully) attempt the *Test your understanding* quiz at the end of each tutorial. You can consider this *training*, because you are *learning* for the weekly quiz.

Then when you sit the weekly quiz, you are being *tested* on *new test* questions. We don't repeat any of the training questions, because you could simply memorise the answers and we want to see how well you have understood the content, not how well you can memorise.

We will be using this terminology during the tutorial. And at the end we'll look at how to divide our data into a training and validation set.

Train-test split

The `sklearn` library provides a useful function that allows us to do this: `train_test_split()`. You will first need to import it with the following:

```
from sklearn.model_selection import train_test_split
```

This function allows the data to be split **randomly** according to some size ratio.

The syntax is:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size = test_size,
                                                    random_state = seed)
```

where:

- `test_size`: the ratio of available data you want to use for the test set (between 0 and 1)
- `random_state`: a seed that ensures the random split will be the same each time

Question! When have we used seeds before?

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

reviews = pd.read_csv('/course/data/Reviews.csv')

X = reviews['Review'].to_numpy()
y = reviews['Rating'].to_numpy()

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

print(X_train.shape)
print(X_test.shape)
```

Note: the function works on both NumPy arrays and DataFrames - if you do use DataFrames just remember to convert to NumPy arrays if you are using `sklearn` models

Question! In the above example we set `test_size = 0.2`. Is this consistent with the shapes of the training and test data?

☆ Split data for SQFT squared

Using the new DataFrame that you created in the previous challenge split the data into a training set and testing set using the `train_test_split()` function.

Use a `test_size` of `0.4` and set `random_state = 1`.

The features we will be using are `SQFT` and `SQFT^2` to predict `Price`.

Print:

- the `shape` of your training features (X)
- the `shape` of your training labels (y)

Your output should look like this:

```
(XXX, X)
(XXX,)
```

Hint! Remember to convert things to NumPy arrays - this will make the next challenge easier

☆ Polynomial regression

Use your training and test data obtained from the previous challenge to train your new regression model:

$$\text{Price} = \beta_0 + \beta_1 \text{SQFT} + \beta_2 \text{SQFT}^2$$

Use `sklearn`'s `LinearRegression` model to predict house prices using the SQFT and SQFT^2 area from the *BatonRouge.csv* dataset.

Extract your β parameters using `.intercept_` and `.coef_` and `print` these to **2** decimal places.

Your output should look like this:

```
beta 0: XXXXXX.XX
beta 1: -XX.XX
beta 2: X.XX
```

Evaluation metrics

In machine learning there are many metrics you can use to determine how well your model predictions agree with data. These metrics often form the basis of objective (loss) functions that can be used to train your model.

Mean Squared Error:

When it comes to regression, it is common to use an objective function based of the mean squared error (MSE). In fact you have already encountered this when doing Ordinary Least Squares.

The MSE is given by the formula:

$$MSE = \frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{N}$$

As you can see from the equation, the MSE looks at the **average squared difference between the predicted value and true value**. Another way to put this is that the MSE looks at the average squared error of each data point.

It will always be **positive (or zero)** as we are taking a sum of squares - values closer to zero are more desirable but be careful not to **overfit**!

Other Metrics

Aside from the MSE there are other metrics you can use to assess the accuracy of your model predictions. Some other metrics that are based on the MSE include:

- Root Mean Squared Error - square root of the MSE

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{N}}$$

- Mean Absolute Error - sum of absolute values of differences instead of the sum of squares

$$MAE = \frac{\sum_{i=1}^{N} |y_i - \hat{y}_i|}{N}$$

Mean squared error

Recall the equation to calculate the mean squared error is:

$$MSE = \frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{N}$$

Calculating MSE by hand

Let's calculate the MSE by hand!

Suppose we had the following true data (y) and predicted data ($\hat{y}$):

$$y = [2,\ 3,\ 1]$$

$$\hat{y} = [3,\ 3,\ 2]$$

The MSE will then be equal to:

$$MSE = \frac{1}{3}\left[(2-3)^2 + (3-3)^2 + (1-2)^2\right] = 0.67$$

The effect of an outlier on the MSE is explored in this supplementary exercise.

Calculating MSE in Python

To calculate the MSE in Python we can use the `mean_squared_error` function from `sklearn.metrics`. We can import it with the following:

```
from sklearn.metrics import mean_squared_error as mse
```

The function is `mse()` and it has the syntax:

```
mse(true_values, predicted_values)
```

Note: you can switch the order of the arguments for the `mse()` function because the MSE involves the difference squared so order doesn't matter

```
from sklearn.metrics import mean_squared_error as mse

y_true = [2, 3, 1]
y_pred = [3, 3, 2]

print(mse(y_true, y_pred))
```

Question! Do you get the same result as when you performed the hand calculation?

☆ Calculate the MSE

Take the model you trained on the **training data** in the previous challenge and *evaluate* it on the **test data** you made during the split.

Calculate the MSE between the test data and your model predictions.

Print the answer to **2** decimal places.

Your output should look like this:

```
MSE: XXXXXXXXX.XX
```

Question! Why is the MSE so high? What can we do to make it more reasonable?

Overfitting and underfitting

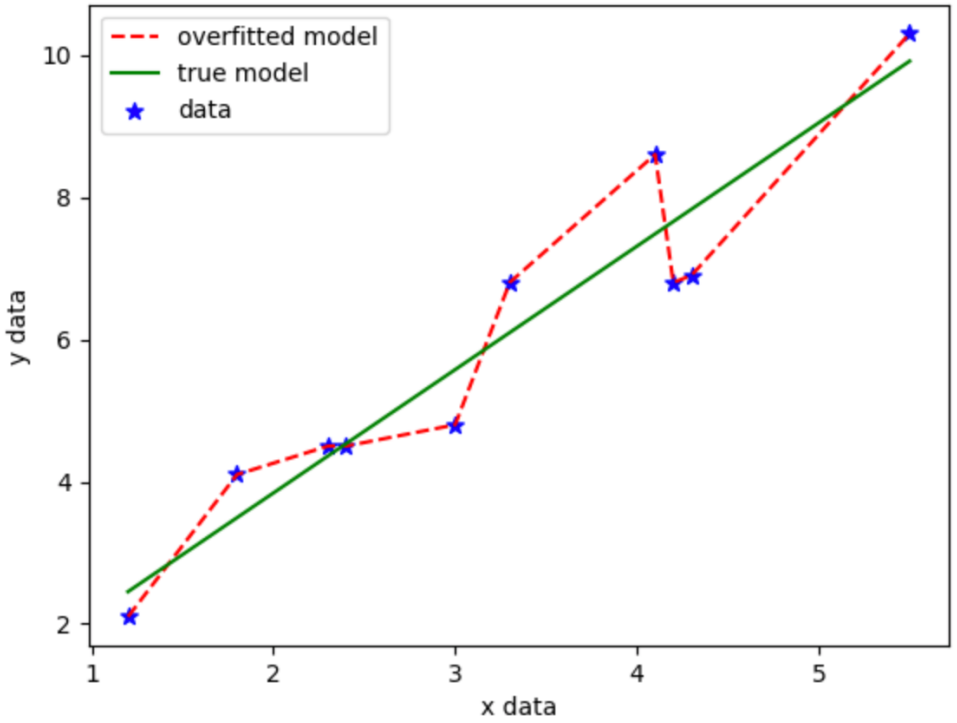
Overfitting (high variance)

Overfitting occurs when you train your model so that it achieves a very high accuracy on the training data, but does poorly on new data. In this scenario, the model is memorising the training data instead of learning the trends and features in the data.

This is bad because every data sample has some unique level of variation due to statistical noise. By memorising the training data, the model fits to the *noise* rather than the features.

When the model is then applied to new data - which has its own statistical noise - the model will then be **unable to generalise to the new data** and it will perform very poorly compared to the training data.

Let us look at an example:



In the overfitted model, the error is zero and the model has learned to perfectly predict all the labels in the training data. However the overall linear relationship in the data has been missed. This is an **extreme case** of overfitting.

Preventing Overfitting

There are many ways to prevent overfitting and this will depend on your model.

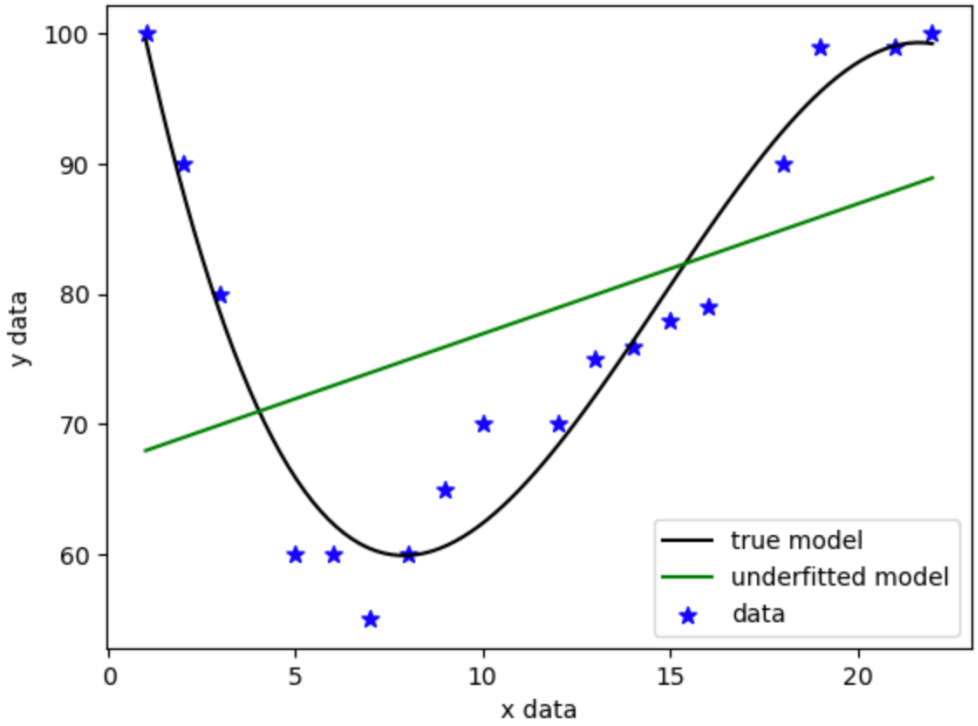
However, there are still fundamental techniques that can be used by all algorithms to help detect and prevent overfitting. These include:

- Holdout (You will learn this today!)
- Cross validation
- Stopping training early once a certain score/error has been achieved
- Regularisation
- Removing features at random
- Increasing the number of training samples

Underfitting (high bias)

The opposite of overfitting is underfitting and this happens when the model is **not complex enough** to learn all the patterns in the data. This happens when the model hasn't been trained long enough or when it **doesn't have enough parameters** to completely describe all the features.

Often underfitting manifests as a relatively low score/high error when training your model and so it is important to ensure that you keep track of the error during training to ensure that you don't overfit or underfit.



In the above example, using a linear model to describe a non-linear relationship leads to underfitting because it is clear that linear terms are not complex enough to describe these relationships.

Hyperparameters

Model parameters

- Model parameters are the values a model *learns* during training.
- The model parameters contain the information required to predict or classify new data.
- Example of model parameters:
    - The β parameters determined during linear regression

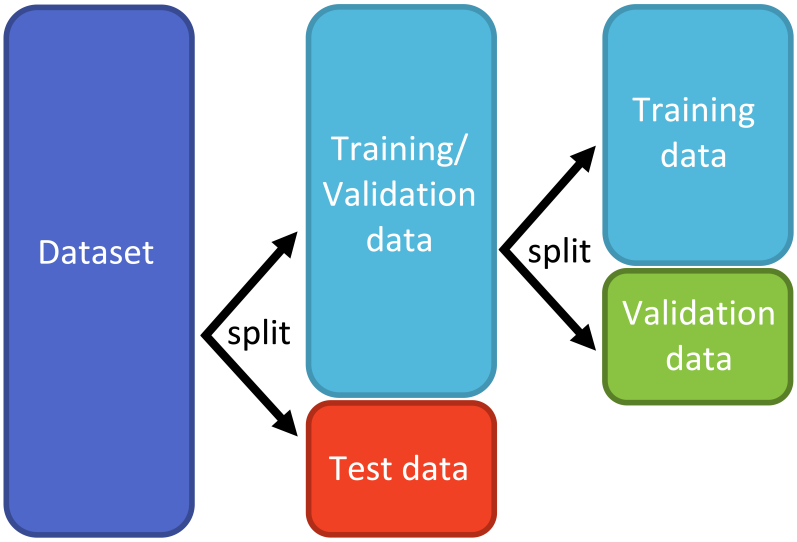
Hyperparameters

- Hyperparameters are parameters which *contro*l how a model will learn during training and must be *assigned prior* to training.
- Hyperparameter *tuning* is an important process used to determine which hyperparameters will result in *optimal performance*.
- Example of hyperparameter:
    - the number of clusters k in clustering

Choosing the right value of hyperparameters can significantly affect the performance of your model - countless research hours are dedicated to finding optimal hyperparameter values for various machine learning models.

Holdout

Holdout is a simple method which will allow you to select optimal *hyperparameters*. To perform holdout, we need to divide our data into 3 datasets: **training**, **validation** and **test** set.



Performing holdout

1. Select an initial (sensible) value for the hyperparameter
2. Train your model using the **training data**
3. Test your model on the validation data with a range of different hyperparameters. Pick the set of hyperparameters which gives you the best performance on the **validation data**
4. Once the optimal hyperparameter values have been chosen, the model is trained on the **combined training and validation data** before being evaluated on the **test data**.

Splitting up the data

An important question is always how to split the available data into 3 smaller datasets. Should they be split equally or should one of the datasets be larger?

The answer often depends on the type of model you have and the amount of available data:

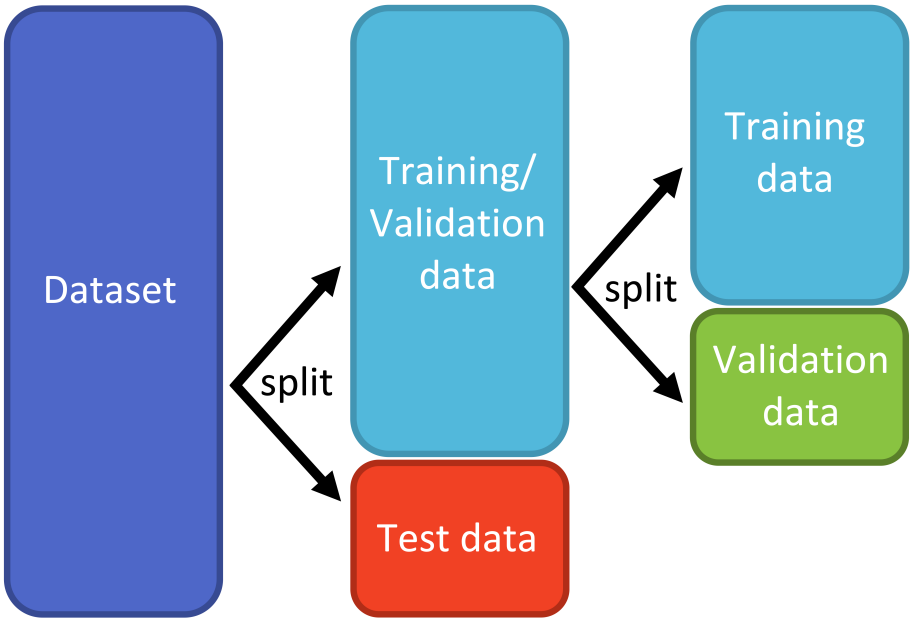
- If you have a model with not many parameters, then making your training dataset smaller might be a good idea
- If you have a large amount of data but a complex model with many hyperparameters then increasing the training and validation datasets would be useful

This decision is up to you and like many others, it's a process of trial and error. A good rule of thumb to use is: 60% for training, 20% for validation and 20% for testing.

Train-vali-test split

In order to perform holdout, we will need to divide our data into a training, validation and test set. We can do this using `sklearn`'s `train_test_split` twice!

The **first** split will create our training/validation data and our **test** data. The **second** split will create our **training** and **validation** data.



Test size

Suppose we want to split out data so that we have:

- 50% training data
- 25% validation data
- 25% test data

It may be tempting to split the data twice, both times setting `test_size = 0.25`. But this won't work. Run the code below and see what happens! The Reviews dataset has exactly 1000 rows.

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

reviews = pd.read_csv('/course/data/Reviews.csv')
X = reviews['Review']
y = reviews['Rating']

X_tv, X_test, y_tv, y_test = train_test_split(X, y, test_size = 0.25)

print('Test:', X_test.shape)
print('Train and vali:', X_tv.shape)
```

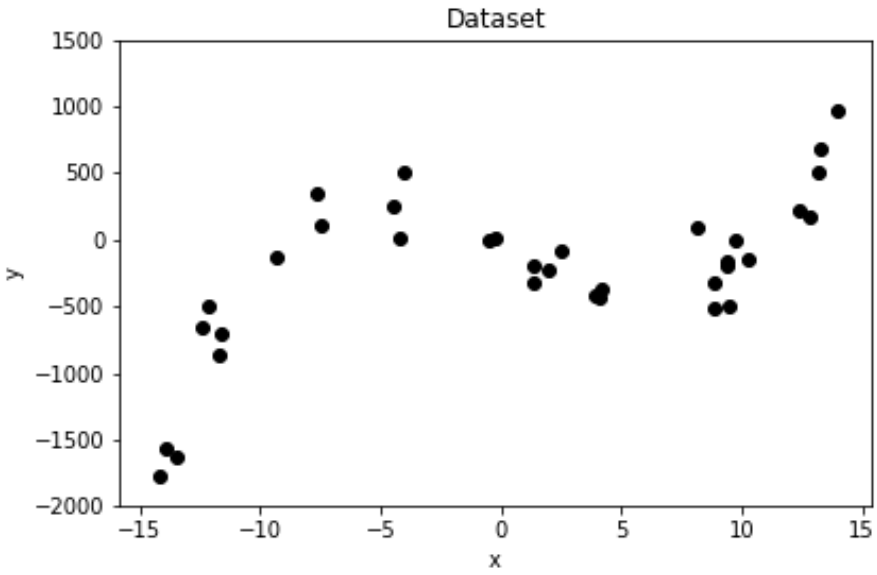
```
X_train, X_vali, y_train, y_vali = train_test_split(X_tv, y_tv, test_size = 0.25)

print('Training:', X_train.shape)
print('Validation:', X_vali.shape)
```

Question! What should we set as our value for the second `test_size` so that we achieve our desired split sizes?

Holdout with polynomial regression

Over the next few challenges, you will use *holdout* to evaluate which polynomial model provides the best fit!

You will be provided with the following data in the file ***data.csv***.



You will fit a polynomial regression model of different orders (p) to the *training data*. In this example, we will only go up to $p = 20$

$$f\left(x\right) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + ... + \beta_{20} x^{20}$$

We will use holdout with the validation data to determine the best polynomial degree.

Question! What would you predict n should be?

Data format

Columns corresponding to each of the features have already been made for you:

```
import pandas as pd

data = pd.read_csv('/course/data/data.csv')
data.info()
```

The simplest thing to do is to convert this into a numpy array:

```
import pandas as pd

data = pd.read_csv('/course/data/data.csv').to_numpy()
```

```
print(data)
```

This will create a matrix with in the following form:

$$\begin{bmatrix} y & x & x^2 & x^3 & \cdots & x^{20} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \end{bmatrix}$$

☆☆ Plot the training data

Write a program that reads in the file *data.csv* into a `numpy` array.

Recall that the data matrix will be in the form:

$$\begin{bmatrix} y & x & x^2 & x^3 & \cdots & x^{20} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \end{bmatrix}$$

where the `0`th column contains the y values and the `1`st column onwards contains the X values.

Extract out the features (X) and target (y). You will then need to break the data up into a training, validation and test set in the following proportions:

- 50% training data
- 25% validation data
- 25% test data

You will need set `random_state = 1` each time you use `train_test_split`.

Generate the scatter plot of the x (only the column corresponding the feature x i.e. not x^2, x^3, ...) and y values of the ***training*** data.

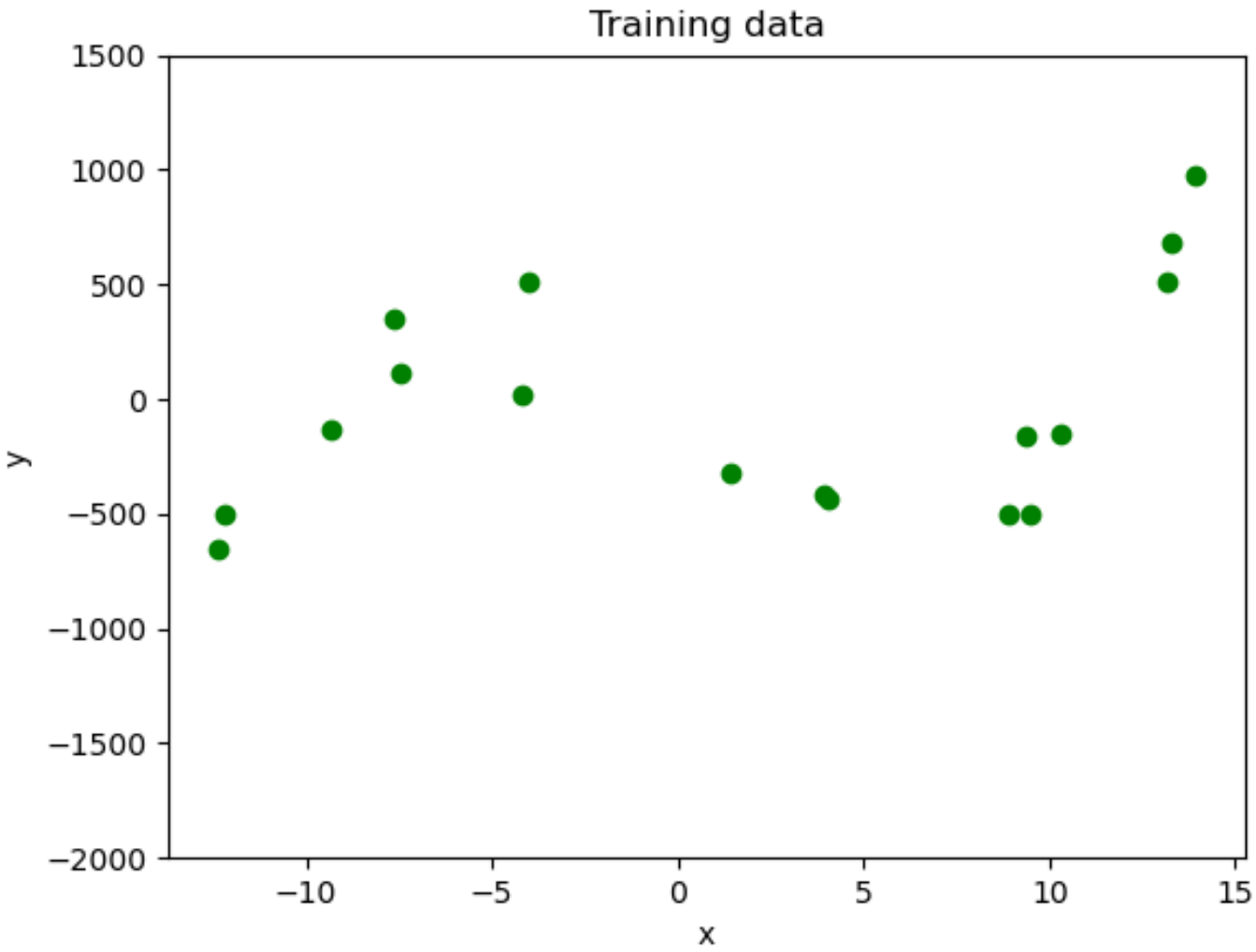
Format your plot so that the markers are `green`.

Code has been provided to ensure the y values only span from -2000 to 1500.

Label your plot with the following:

- x axis label: *x*
- y axis label: *y*
- title: *Training data*

Your plot should look like this:


Training data
1500
1000
500
0
−500
−1000
−1500
−2000
y
−10
−5
0
5
10
15
x

☆ Polynomial degree 20

Copy and paste your code from the previous challenge where you create your training, validation and test data.

Then write a program that uses `sklearn`'s `LinearRegression` model to predict y from the x values following the equation:

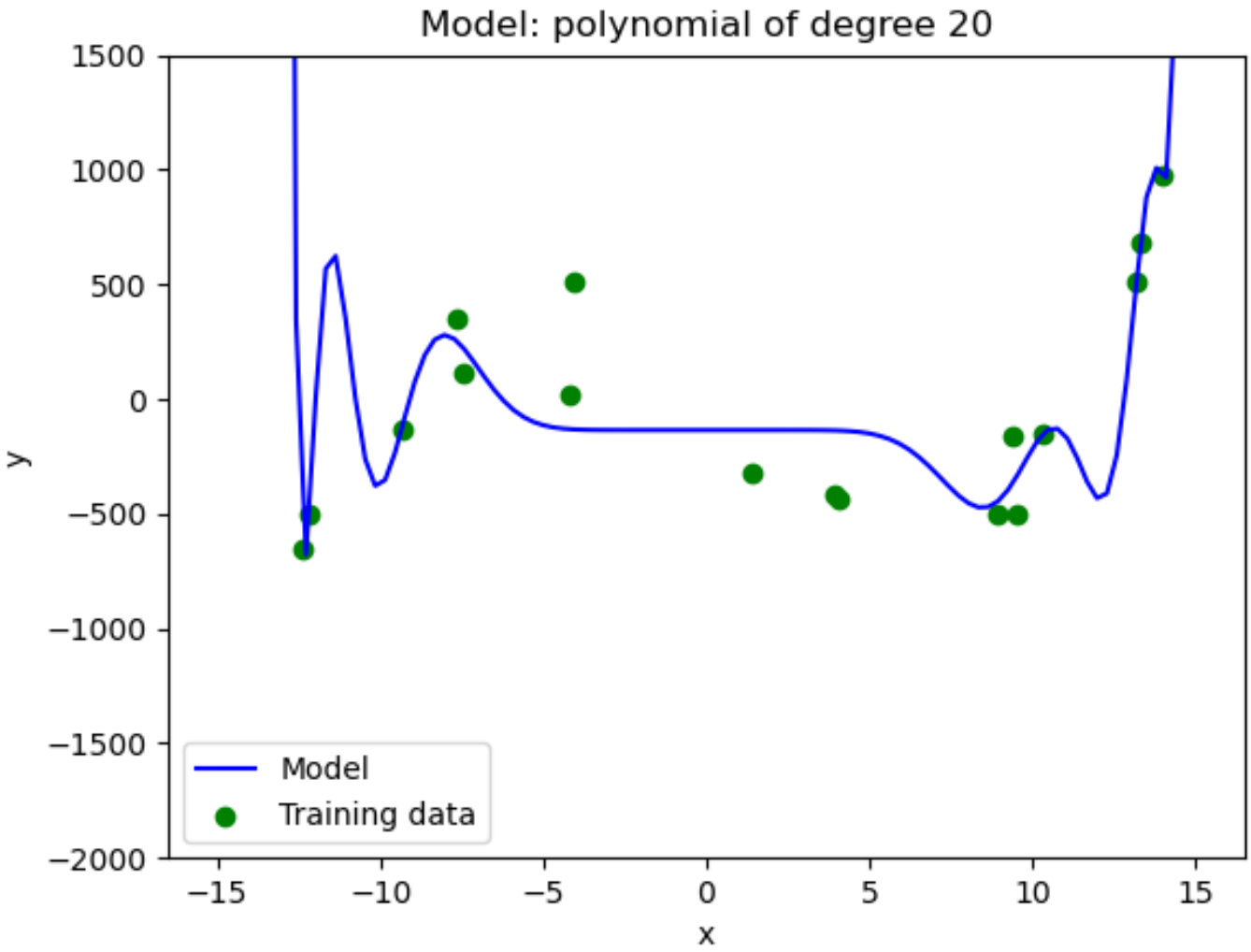
$$f\left(x\right) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + ... + \beta_{20} x^{20}$$

The code to plot your model has already been provided. This will format your plot so that the markers for the training data are **`green`**, the linear regression model is **`blue`** and the y values only span from -2000 to 1500.

It also labels your plot with the following:

- x axis label: *x*
- y axis label: *y*
- title: *Model: polynomial of degree 20*
- training data: *Training data*
- model: *Model*

Your plot should look like this:

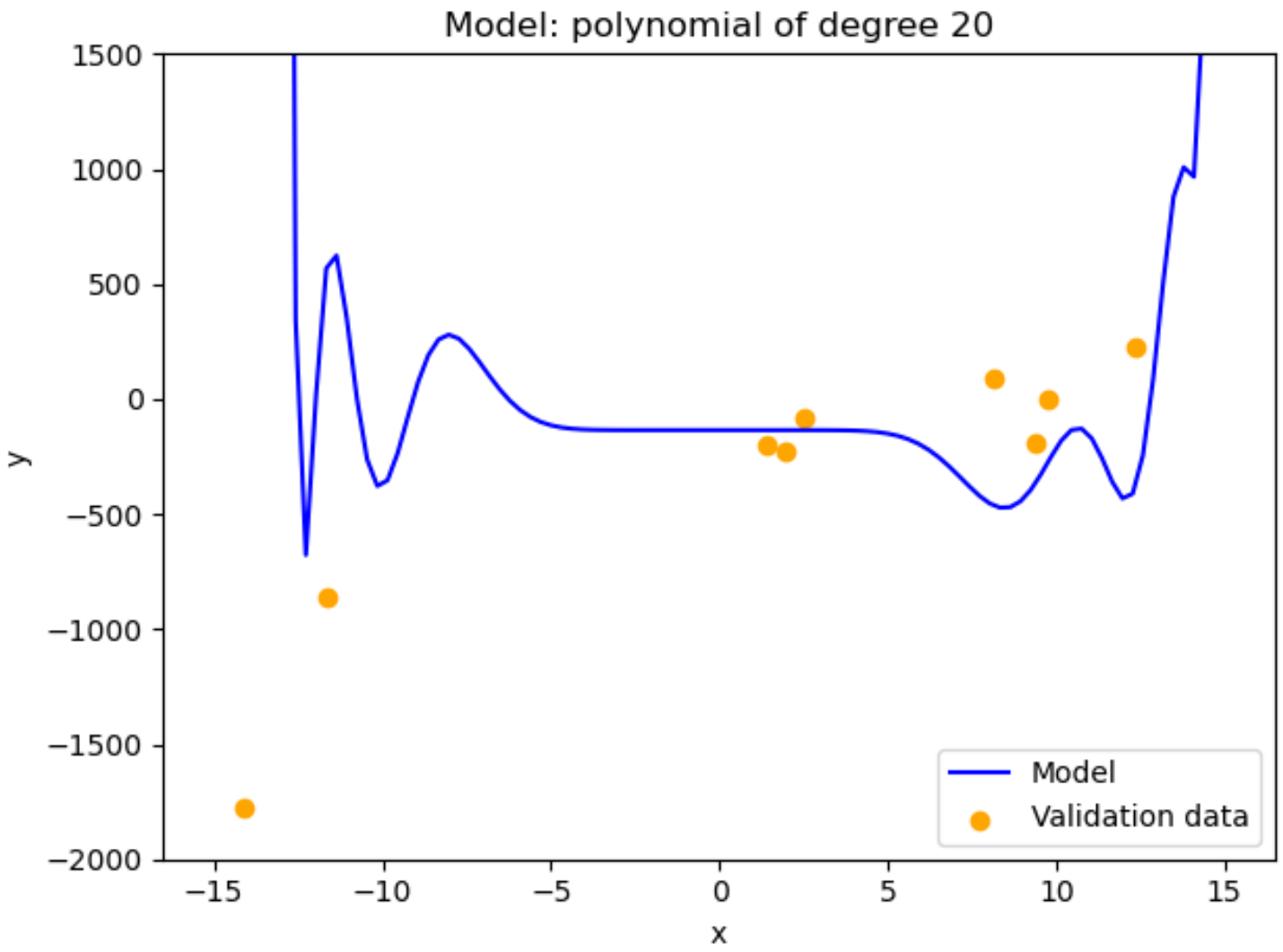

Model: polynomial of degree 20
1500
1000
500
0
−500
−1000
−1500
−2000
y
−15
−10
−5
0
5
10
15
x
Model
Training data

☆ Visualise the model on validation data

Now we will visualise how good our model is on the validation data.

Copy and paste in your code from the previous challenge and modify it to plot only your model with the validation data. Format your plot so that the markers for the validation data are `orange`.

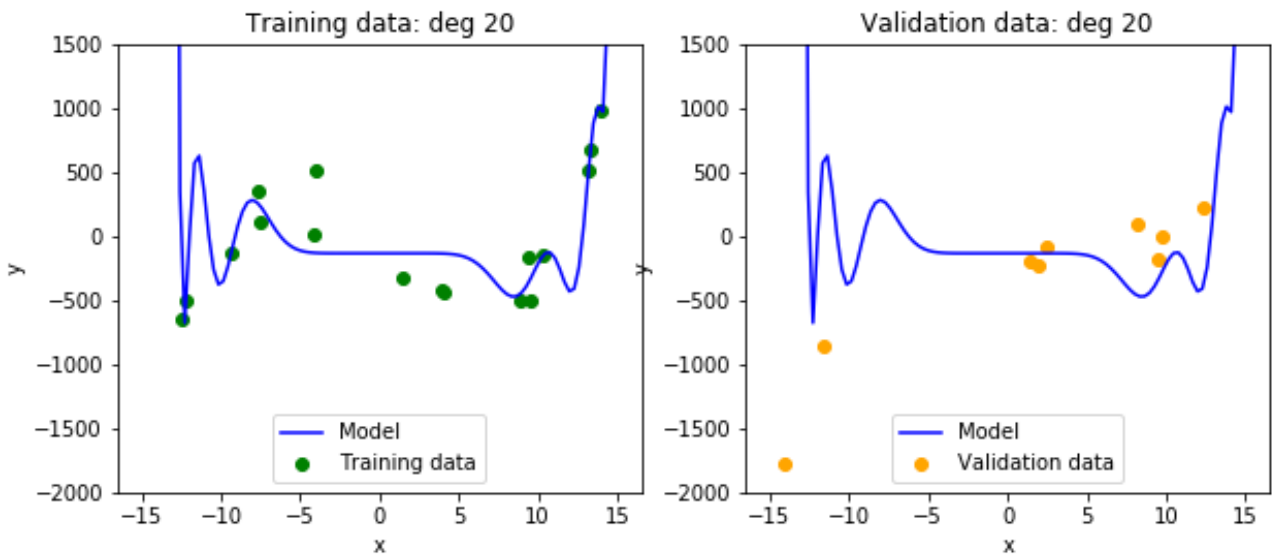
Your plot should look like this:



Question! Do you think our regression model fits the validation data well? Why or why not?

☆ Calculate the MSE

We have now observed our linear regression model on our training and validation data.



To evaluate our model on our training and validation data we will calculate the mean squared error.

Complete the scaffold provided. This will `print` out your results to **4** decimal places. Your output should look like this:

```
Train mse: XXXXX.XXXX
Validation mse: XXXXXXXXXX.XXXX
```

Question! Do you obtain a better mean squared error on the training data or the validation data?

✏☆☆☆ Holdout validation

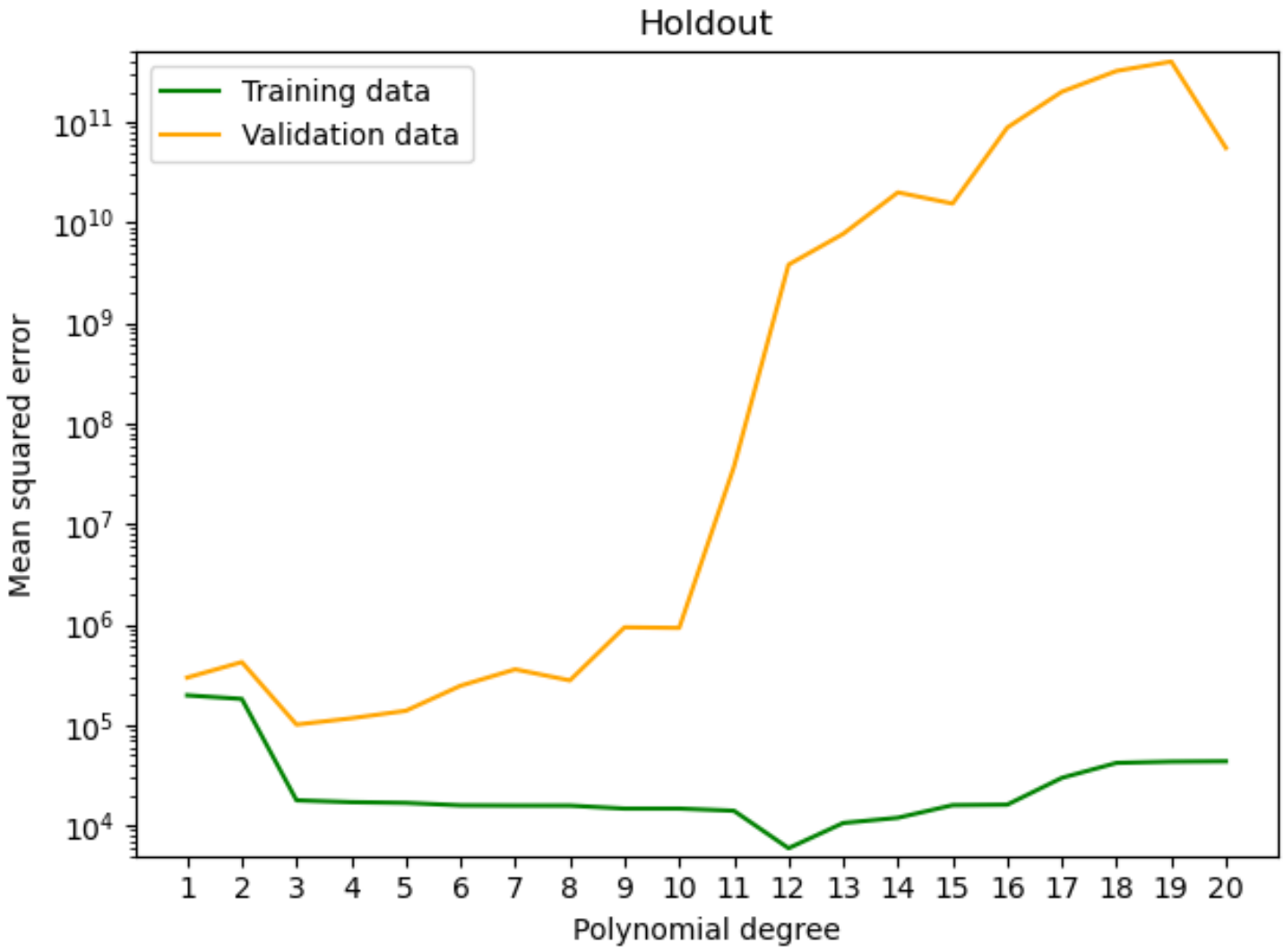
In the previous challenge you calculated the mean squared error on the training and validation data for a linear regression model of degree 20.

In this challenge you will need to do this again for degrees 1 to 20.

Fill in the scaffold with code from your previous challenge. You will need to modify it so that at each iteration you fit a polynomial of degree `deg` and save the mean squared errors in `mse_train` and `mse_vali`.

The code to plot this has already been provided. This will format your plot so that training mse is in `orange` and the validation mse is in `green`. It will also put your y axis on a log scale and set the limits to $(5 \times 10^3, 5 \times 10^{11})$.

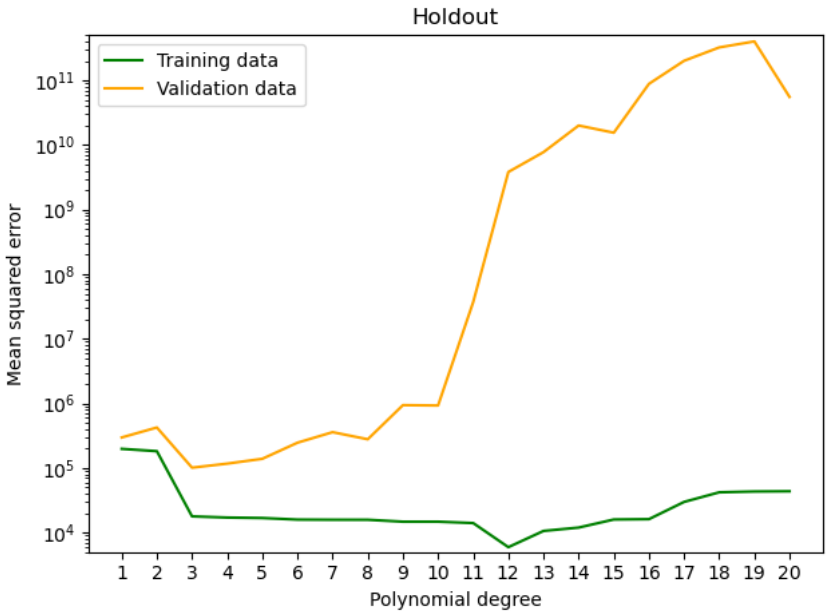
Your plot should look like this:



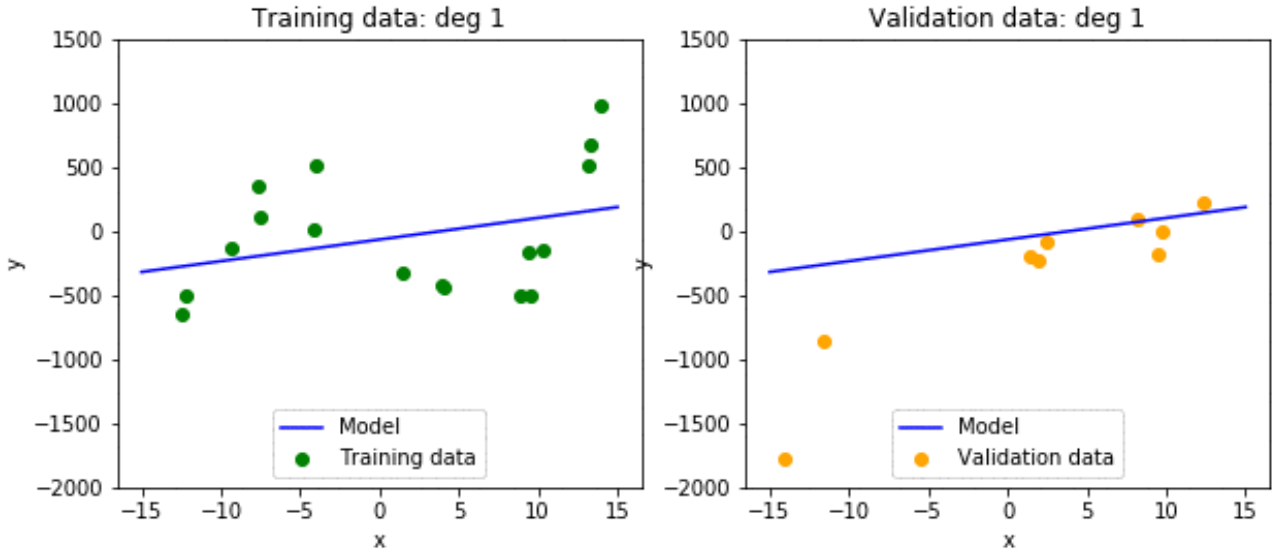
Question! You can see that to generate this plot we use `plt.yscale('log')`. What does this do?

Interpreting the results

Here is the figure you produced from the previous challenge:



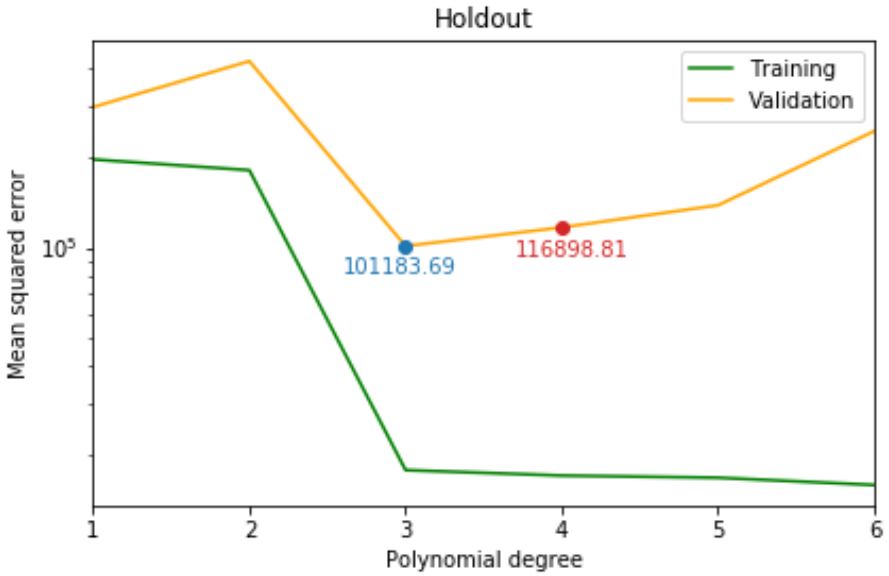
Let's have a look at the different fits!



Question! Which of the models do you think fits our data the best?

Closer look

Let's have a closer look at our holdout diagram.



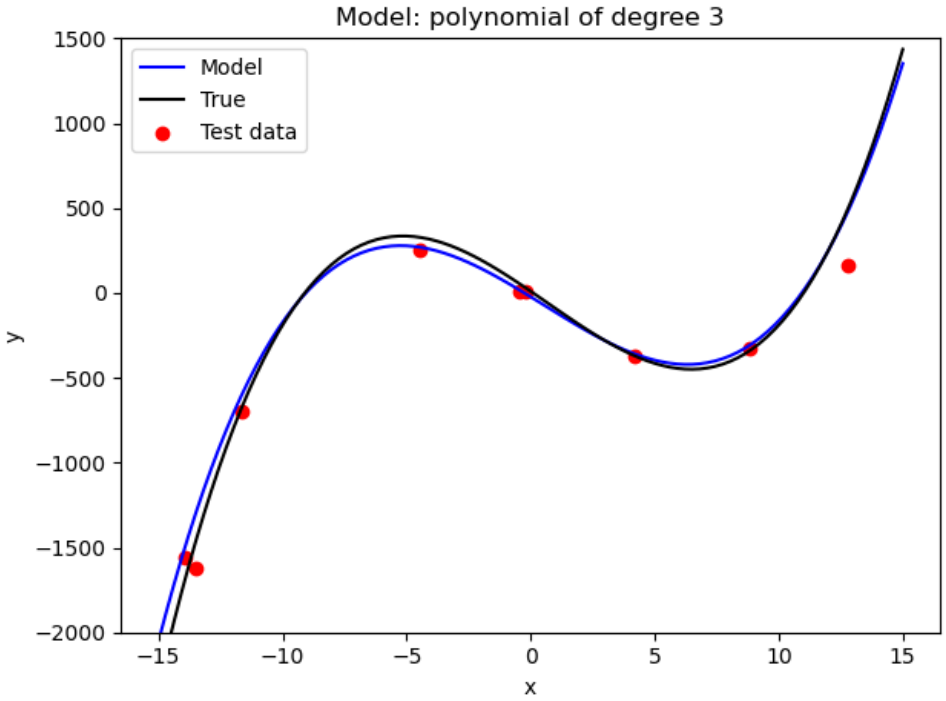
Question! What is the polynomial degree that corresponds to the *minimum* validation MSE? What should we then select to be our polynomial degree?

✏☆☆ Polynomial degree 3

Copy and paste your code from the *Polynomial degree 20* challenge. Modify the code so that you fit a polynomial of degree 3 on the **combined training and validation data**.

You have been provided with code that plots your model. **Do not delete this code.** You will also need to code to plot the **test** data. Format your plot so the makers are `red`.

Your plot should look like this:



✏☆ Evaluate your model

Take the model you trained in the previous challenge. Evaluate your model on test data.

Complete the scaffold provided. This will `print` out your results to 4 decimal places. Your output should look like this:

```
Test MSE: XXXXX.XXXX
```

Summary

Fabulous work! Stop and appreciate about how much you have learnt!

Polynomial Regression

- Polynomial regression allows us to build a ***nonlinear*** model.
- These models can include terms of a degree greater than 1

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + ... + \beta_n x^n + \epsilon$$

- They can also include new features created by multiplying together original features. E.g. x_2^2 or $x_1^2 x_4$
- Polynomial regression is a **special case** of linear regression

Overfitting and underfitting

- **Overfitting (high variance):** The model is overly complex memorises the data and is unable to generalise to new data.
- **Underfitting (high bias):** The model is not complex enough and is unable to learn all the trends in the data.

Parameters

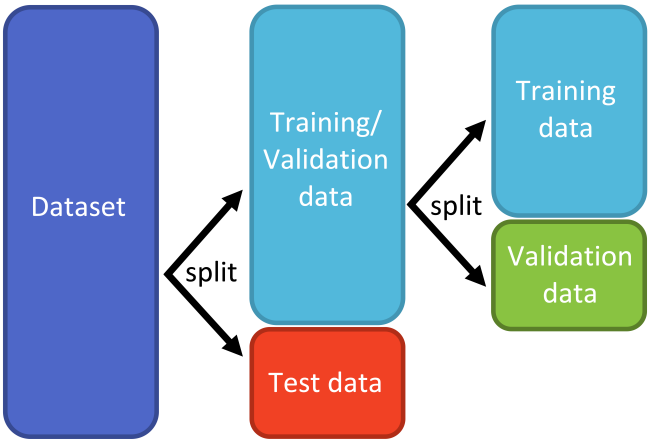
- **Model parameters:** What your model learns during training e.g. β parameters during regression
- **Hyperparameters:** Control how your model will learn during training and must be assigned prior to training. e.g. k in k-means clustering.

Holdout

A method for selecting optimal hyperparameters.

- **Training data:** data your model *learns* from during the `.fit()` process.
- **Validation data:** data you use to determine the best *hyperparameters*. You will need to use `.fit()` and `.predict()` on the validation data.
- **Test data:** data you use to *evaluate* your model when you use a `.predict()`

Train-vali-test split



To split the dataset into train, validation and test sets you can use `sklearn`'s `train_test_split()`.

```
from sklearn.model_selection import train_test_split

X_tv, X_test, y_tv, y_test = train_test_split(X, y,
                                              test_size = test_size,
                                              random_state = seed)

X_train, X_vali, y_train, y_vali = train_test_split(X_tv, y_tv,
                                                    test_size = validation_size,
                                                    random_state = seed)
```

Evaluation Metrics

- Mean square error (MSE)

$$MSE = \frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{N}$$

```
from sklearn.metrics import mean_squared_error as mse

mse(predicted values, true values)
```

Tutorial Feedback Survey

Please give the teaching team some feedback for this week!

Question 1

How did you feel after the tutorial?

- ○ Great!
- ○ Satisfied
- ○ Disappointed
- ○ Confused

Question 2

How do you feel about the speed of the tutorial?

- ○ Too fast
- ○ Just right
- ○ Too slow

Question 3

How do you feel about the difficulty of concepts?

- ○ Too hard
- ○ Just right
- ○ Too easy

Question 4

The material was clear and easy to understand

- True
- False

Question 5

The tutor was clear and easy to understand

- True
- False

Question 6

Please write any general comments, feedback or elaborate on your previous answers.

No response

Test your understanding

Question 1

Why is overfitting our models bad?

- Overfitting leads to high bias models which is not desirable
- Overfitting requires too much computational resources for it to be a feasible way of training our models
- Overfitting prevents our model from generalising well to new data
- Overfitting is actually good because it leads to minimal errors

Question 2

Is this statement True or False?

> An overfitted model will have high bias

- True
- False

Question 3

What are the steps for holdout? Reorder the steps so that they are in the correct order for the holdout process.

Repeat steps 1 and 2 for different hyperparameters:

> 1.
> 2.

3.
4.

5.

- Choose the best hyperparameters based on the model's performance on the validation data
- Train your model using the training data
- Train your model on the combined training and validation data
- Evaluate your model on the test data
- Evaluate your model on the validation data

Question 4

You believe the polynomial regression model that will best fit your data has the equation:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \beta_4 x_2^2$$

You have the following 4 data points:

$$X = \begin{bmatrix} 12 & 2 \\ 5 & 5 \\ 9 & 3 \\ 3 & 8 \end{bmatrix}$$

You want to use `sklearn`'s `LinearRegression` to fit your model:

```
linear_reg = LinearRegression()
linear_reg.fit(X, y)
```

Which of these should be your matrix **X** that you use in `.fit()`?

- $$X = \begin{bmatrix} 12 & 2 \\ 5 & 5 \\ 9 & 3 \\ 3 & 8 \end{bmatrix}$$
- $$X = \begin{bmatrix} 12 & 2 & 144 & 4 \\ 5 & 5 & 25 & 25 \\ 9 & 3 & 81 & 9 \\ 3 & 8 & 9 & 64 \end{bmatrix}$$

$$X = \begin{bmatrix} 12 & 2 & 24 & 4 \\ 5 & 5 & 25 & 25 \\ 9 & 3 & 27 & 9 \\ 3 & 8 & 24 & 64 \end{bmatrix}$$

$$X = \begin{bmatrix} 12 & 2 & 24 \\ 5 & 5 & 25 \\ 9 & 3 & 27 \\ 3 & 8 & 24 \end{bmatrix}$$

Question 5

You have built a linear regression model to predict the following target values:

$$\text{true labels} = [4, 1, 0]$$

Your model makes the following predictions:

$$\text{predicted labels} = [2, 3, 1]$$

What is the MSE?

- 7
- 1.67
- 5
- 3

☆ Train test split for Reviews

Write a program that reads in the file *Review.csv* into a `pandas` DataFrame then extracts out the Review (X) and Rating (y) variables. You will then need to break the data up into a training, validation and test set in the following proportions:

- 60% training data
- 20% validation data
- 20% test data

Print the `shape` of:

- Your training features
- Your validation features
- Your test features

Your output should look like this:

```
Training: (XXX,)
Validation: (XXX,)
Test: (XXX,)
```

☆ MSE with outliers

In this task you will need to calculate two different MSE values.

Calculate the MSE of the following predicted values given the true values:

```
pred_vals1 = np.array([3.2, 7.0, 4.2, 7.4, 3.2, 4.8, 5.1, 6.9, 1.9])
true_vals1 = np.array([2.9, 6.5, 4.4, 8.1, 3.1, 4.3, 5.6, 6.2, 2.5])
```

Then, calculate the MSE of the following new predicted values given the new true values:

```
pred_vals2 = np.array([3.2, 7.0, 4.2, 7.4, 3.2, 4.8, 5.1, 6.9, 1.9, 6.8])
true_vals2 = np.array([2.9, 6.5, 4.4, 8.1, 3.1, 4.3, 5.6, 6.2, 2.5, 13.7])
```

You will notice that the only difference between the two datasets is that there is an outlier in the second dataset.

Print your answers to **4** decimal places. Your output should look like this:

```
MSE of Data 1: X.XXXX
MSE of Data 2: X.XXXX
```

Question! Why are the two MSE values so different when only one data point has been added?

✏☆☆☆ OLS for polynomial regression

Using OLS, find the optimal β parameters for the following polynomial regression model:

$$\text{Price} = \beta_0 + \beta_1 \text{SQFT} + \beta_2 \text{SQFT}^2$$

You will first need to create the SQFT^2 column, the same way you did in the SQFT squared feature challenge.

You will want to fit your model on the *training* data only. Break your data up into a training and test set by setting `test_size =0.4` and `random_state = 1`.

Now use OLS! Recall the the OLS equation is:

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Print out the β values to **2 decimal places**. Your output should look like this:

```
beta 0: XXXXXX.XX
beta 1: -XX.XX
beta 2: X.XX
```

Question! How do these values compare to those you found when training your model in the *Polynomial regression challenge*?

☆☆ Adding SQFT cubed

Consider the following polynomial regression model:

$$\text{Price} = \beta_0 + \beta_1 \text{SQFT} + \beta_2 \text{SQFT}^2 + \beta_3 \text{SQFT}^3$$

Modify the *BatonRouge.csv* data to include **two new columns**: `SQFT^2` and `SQFT^3`.

You will want to fit your model on the *training* data only. Break your data up into a training and test set by setting `test_size =0.4` and `random_state = 1`.

Then find the optimal β values for this model using OLS. Recall the the OLS equation is:

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Print out the β values to **2 decimal places.** Your output should look like this:

```
beta 0: -XXXX.XX
beta 1: XX.XX
beta 2: -X.XX
beta 3: X.XX
```

Question! What does the value of β_3 tell you about the feature SQFT^3?