GENERALISATION

Dr. Brian Mc Ginley

GENERALISATION

- Generalisation is the goal of ML
  - Want good performance for **new** data
  - How good the generalisation, is how well it performs on previously "unseen data" i.e. data not used to train the set.
- **Definition:**
  - Generalisation is the model's ability to give sensible outputs to sets of input that it has never seen before.
- Root-mean-squared (RMS) error could be used to measure: $E_{RMS} = \sqrt{2E(w*)/N}$
- Back to previous idea of checking how far away pred is away from the original y, what is the problem?
- We are checking the Generalisation of the model by using data it was trained on - i.e. it is NOT previously unseen.
- Our model may be **overfit** to the training data.

UNDERFITTING

- The opposite problem is under-fitting.
- You can't even seem to reduce your training loss to anything reasonable. This is like the student not being smart enough to do the exercises.
- In machine learning this is generally a symptom of not having a complex enough model (e.g. student not being smart enough).
- Students not being smart enough is unlikely in the real world, as each of us is given a complex model to work with (the human brain).
- Other causes of under-fitting in machine learning are poor training mechanisms or an inappropriate loss function.
- This is more like the student who sits in the library all day long - looking at social media and can't figure out why they still can't do the examples.

UNDERFIT/OVERFIT

- **Definition:**
- Underfitting occurs when a statistical model or machine learning algorithm cannot capture the underlying trend of the data.
- Intuitively, underfitting occurs when the model or the algorithm does not fit the data well enough. Underfitting is often a result of an excessively simple model.

- **Definition:**
- Overfitting occurs when a statistical model or machine learning algorithm captures the noise of the data. Intuitively, overfitting occurs when the model or the algorithm fits the training data too well.
- Overfitting is often a result of an excessively complicated model.

BUILDING A MODEL

- It does not matter what type of model we are building the process is similar.
  - Have a set of data.
  - Pick a base form of model to use.
  - **Train** the data on that type of model (choosing the hyper-parameters to ”tweak” the model e.g. what size polynomial)
  - **Test** the model.
- Note: no matter how good the model, almost always the more datapoints the better. (The more tuples of **x**, y the better, having too many independent variables/features can cause issues.)

BUILDING A MODEL

- We have a set of data, that is a pair of information. We know the correct answer for the set.
  - X is the set of all inputs (a matrix).
  - y is the set of all corresponding response variables.
  - A particular $y^{(i)}$ is the result for a particular $\mathbf{x}^{(i)}$.

```
from  sklearn.linear_model  import  LinearRegression

lr = LinearRegression()
lr.fit(X,y)
pred = lr.predict(X)
```

TESTING THE PERFORMANCE OF THE MODEL

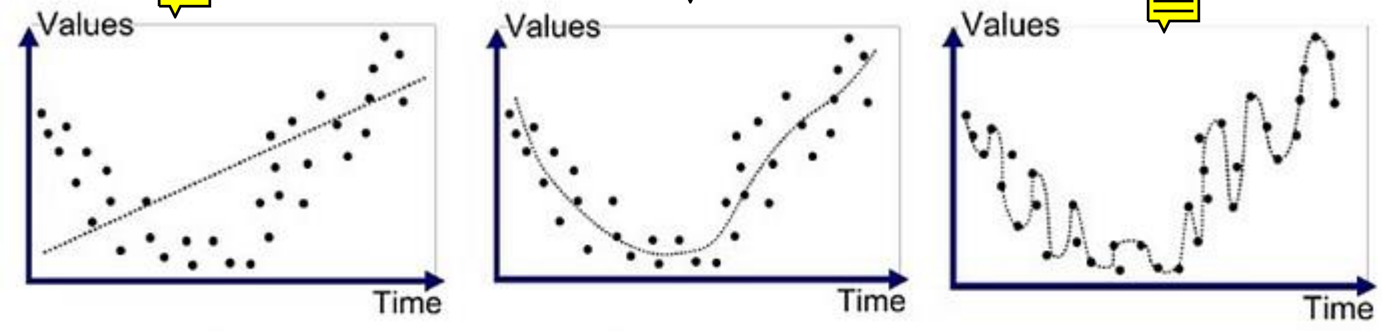
```
lr.fit(X, y)
pred = lr.predict(X)
```

- In the code, we have created a model called *lr*.
  - *lr* was **trained** on the set of *known* information.
  - Recall from regression, the predicted values are not exactly the same as what is inputted
  - So *pred* is not the exact same as *y* but they hopefully are close.
  - So we could test performance by checking how far away *pred* is away from the original *y*
  - Does anyone see a problem with this?

OVERFIT/UNDERFIT REVISITED

- Overfitting is the case where the overall cost is really small, but the generalisation of the model is unreliable. This is due to the model learning "too much" from the training data set. (low bias, high variance)
- Underfitting is the case where the model has "not learned enough" from the training data, resulting in low generalisation and unreliable predictions. (high bias, low variance)

BIAS/VARIANCE



Underfitted

Good Fit/Robust

Overfitted

- Bias: High bias (making a lot of assumptions about the underlying model) can result in underfitting.
- Variance: Sensitivity of the model to the training data
    - More sensitivity = higher variance
- Bias/Variance trade-off

OVERFIT/UNDERFIT REVISITED

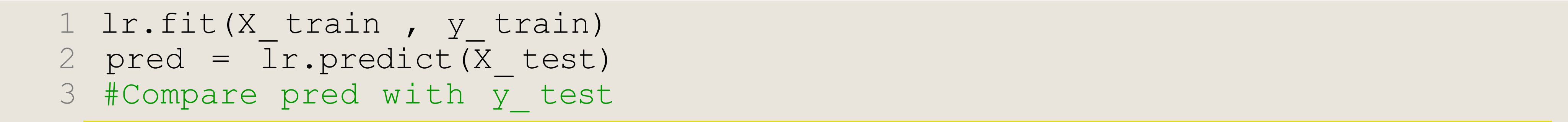
- Bias-variance trade-off. What is the right measure?
  - Depending on the model at hand, a performance that lies between overfitting and underfitting is more desirable.
  - This trade-off is the most integral aspect of Machine Learning model training. Machine Learning models fulfil their purpose when they generalise well.
  - Generalisation is bound by the two undesirable outcomes — high bias and high variance.
  - Detecting whether the model suffers from either one is the sole responsibility of the model developer.

CONFIDENCE

- One of the major problems with over-fitting is that it does not just mis-classify new data, but it can often do so with very high confidence.
- When the model is of more modest complexity, it may have certain samples in the training set that end up on the wrong side of the decision boundary (classification) or are a long way from the model (regression).
- However, the model can report this.
- The model can say, I predict this output, but I am only *x%* sure (classification) or there is x-tolerance with this prediction (regression).

HOW DO WE TEST OUR MODEL

- We use a test set of data, that we know the answers.
- It does not matter if the response variable is a number (regression) or a quality (category/classifier), it's a similar idea.
- We may use different formulae to determine accuracy and performance.
- Our test set is kept separate from the training set.
- When it is used to check performance of the model, it is brand new information to the model - while we still know the correct answers



```
lr.fit(X_train , y_train)
pred = lr.predict(X_test)
#Compare pred with y_test
```

- Compare pred with the known results for the test set and it will give allow us to check performance.

TEST SETS

- Sometimes, when trying to solve a problem using you are given an explicit *Training Set* and a separate *Test Set*.
- The Training set is used to *fit* the model (learn the parameters).
- The Test set is used to *evaluate* the model (check its performance/generalisation).
- However, often you are just given one set of data and it's your job to decide the best method to partition the dataset.

TRAIN/TEST SPLIT

- Let's say we start with a big set of data that we have collected. This is a sample of size m.
  - For each item *i* in the data, we know the $\mathbf{x}^{(i)}$ vector (features/independent variables)
  - We also know the equivalent $y^{(i)}$ response.
  - We split the data into two sets, a training set and a test set
    - Almost always the training set is going to be significantly larger than the test set. 80/20, 75/25 are often good splits.
    - Randomly select which items are in the training set and which are in the test set (**very important**).
  - **Fit** the model on the **training** set.
  - **Evaluate** the model on the **test** set.

TRAIN/TEST SPLIT

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

- is a method built into sklearn to split a dataset into training and test sets randomly.

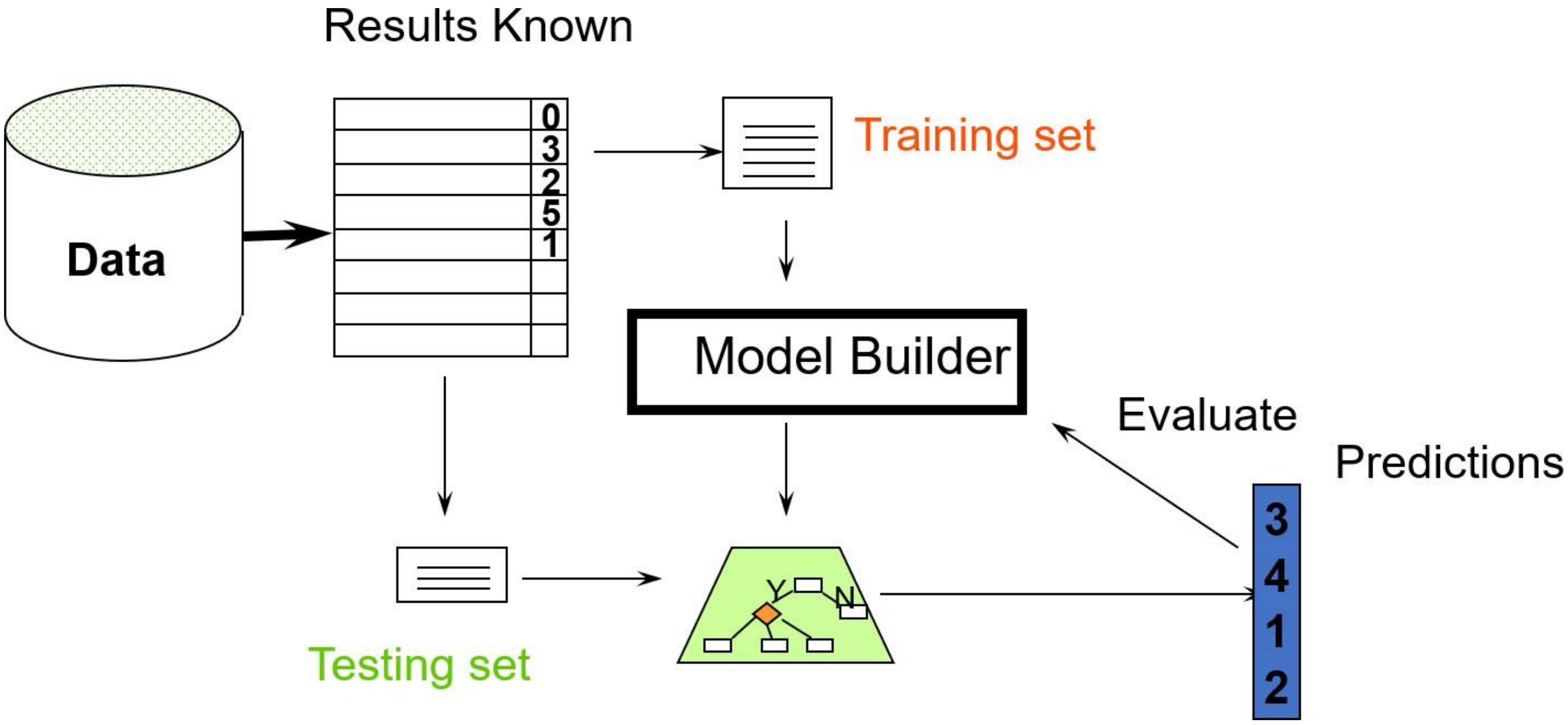
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
```

- will make a test size of 25% of the data (which is the default anyway)

```
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.75)
```

- can specify it that way too if you prefer.
- The return values will be numpy arrays.

FULL MODEL PROCEDURE



TEST SET

- Some Important Notes About the Test Set
  - You need to guard the test set carefully, from yourself, and use it only sparingly.
  - The Test set should be kept apart and almost never used.
  - It must not be used in trying to choose any hyper-parameter (polynomial order (see later)) for a model
    - If it was used in selection, it would no longer be unseen data.
  - Use it only for overall evaluation.
  - ML testing competitions keep the test set hidden and private.

EXAMPLE

- Say we have collected 3000 emails that have been tagged as either spam or not-spam.
  - $\boldsymbol{x}^{(i)}$ stores the number of times a particular word is in the email.
  - Our target set $t = (0, 1)$, 0 for not-spam, 1 for spam.
  - So, each $y^{(i)}$ is either 0 or 1.
  - Split the set into training and test sets.
  - Fit
  - Evaluate
  - What happens if we have many more non-spam vs spam?
  - We need a representative sample i.e. a certain percentage of spam emails, a certain percentage of non-spam emails that are of one particular type, then another percentage of non-spam emails that of are another type etc. etc.

SEEING OVERFITTING/UNDERFITTING

- Often people compare the score for the training set vs the score for the test set to see if either of these are happening

```
lr.fit(X_train, y_train)

train_score = lr.score(X_train, y_train)
test_score = lr.score(X_test, y_test)
```

  - If the training score is very poor, then the model is underfit
  - Almost always the training score will be better than the test score.
  - If there is a large difference between training score and test score (training score much better), than the model is overfit

DEALING WITH OVER/UNDERFITTING

- It is a constant issue but there are some common techniques
- For under-fitting you either need to train for longer or use a more complex model. It may also require you to look at your training mechanisms and loss function. Are they appropriate to the problem?
- For over-fitting you could:
  - Make the model less complex.
  - Decrease the number of features. Particularly if there is a large number of features compared to the number of training samples. But you are also throwing away information with this method. Choose wisely what to throw out.
  - Get more data.
  - Most often, use regularisation (later we'll talk about this).

DECIDING THE COMPLEXITY OF THE MODEL

- When dealing with relatively simple models, (that take only minutes-hours to train), then you can afford to try many different model sizes.
- But you can't just pick the one that gets the lowest training error, we've already shown that can lead us wrong.
- Instead, we need a mechanism to check our loss on data that the training algorithm doesn't see (test set??).
- The technique we will be looking at is called Hyper-parameter tuning.

HYPER PARAMETERS

- **Definition:**
  - Hyper-parameters are parameters that are chosen rather than learned.

- Examples of Hyper-parameters are:
  - the degree of your model, e.g. linear, quadratic, n-degree polynomial. Which features to use in your model? (Is every bit of information needed?)
  - the learning rate α
  - the amount of Regularisation
  - the number of layers/neurons in a Neural Network

POLYNOMIAL CURVE FITTING

- What form is f(x)?
  - Try polynomials of degree M.

$$f(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M$$

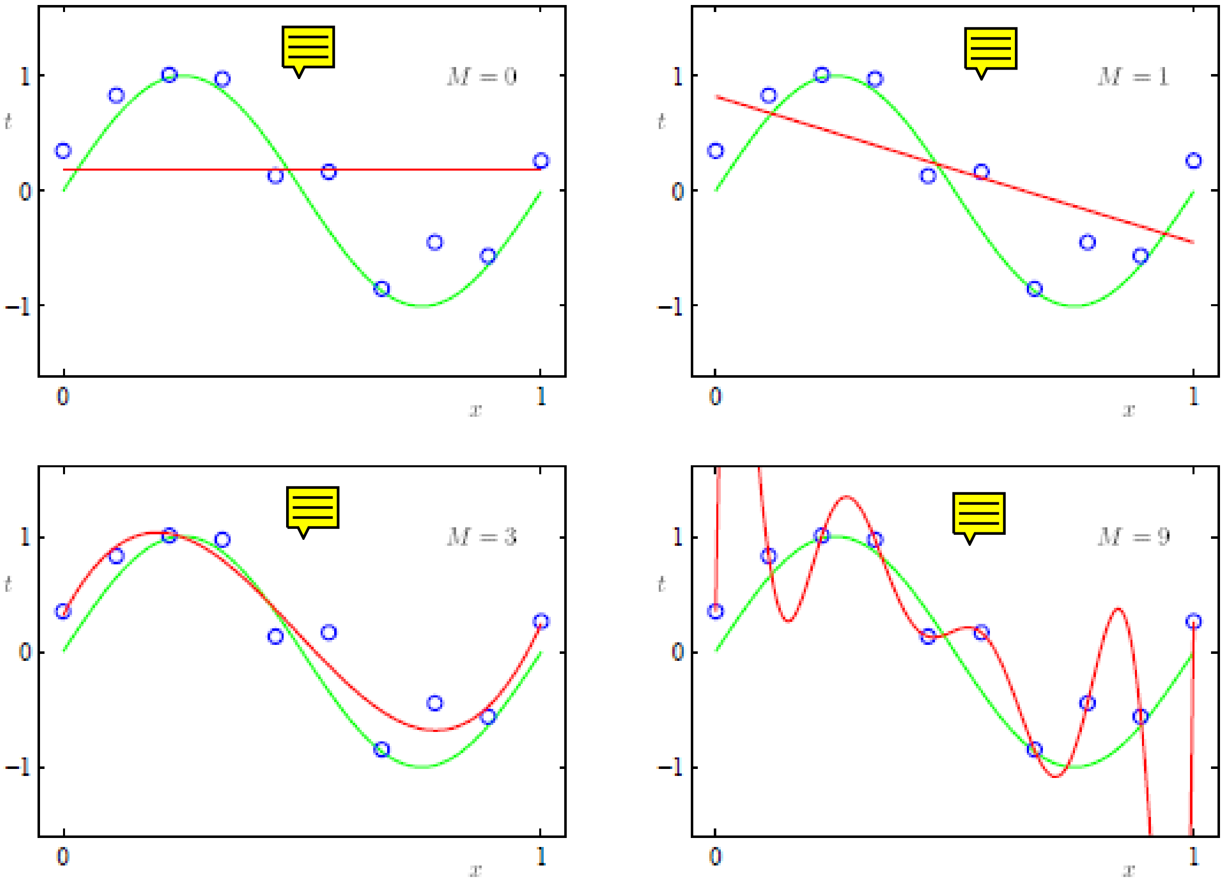
  - This is the *hypothesis space*.
  - Note ***w*** is a vector (that's why bold). $\mathbf{w} = (w_0, w_1, \ldots, w_M)$
- How do we measure success?
  - Sum of squared errors

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} [f(x_n, \mathbf{w}) - t_n]^2$$

  - or RMS
- Among functions in the class, choose degree which minimises this error.
  - The degree of polynomial is a **hyperparameter**.
  - We really want the degree that captures the trend of the data best.

WHICH DEGREE OF POLYNOMIAL?

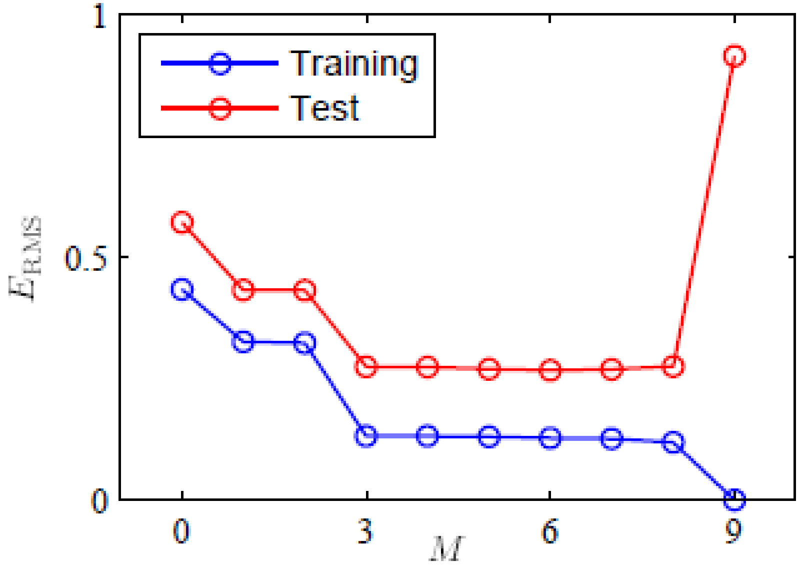
- The green line is the “ideal” model, the red line is the shape of the model built.
  - If $M = 9$ then $E\ (w^*) = 0$: This is over-fitting



RMS ERROR

$$f(x, w) = w_0 + w_1x + w_2x^2 + \ldots + w_Mx^M$$

- Here is the plot of the root-mean-squared for each M for:
  - the performance of the training set vs the model (known information)
  - the performance of the test set vs the model (previously unseen)



- The point is, the model performs very very well at M = 9 for the training set. It performs very very poorly at M = 9 for the test set. i.e. it is overfit at M = 9.

SELECTING HYPERPARAMETERS

- We need to have some method of selecting the hyperparameters.
- Note: There can be many different methods for selecting hyperparameters (not the focus of this module, but 1-method will be demonstrated). In the future, if you go into this area, you will have more learning to do.

VALIDATION SET

- Keeping the Test Data separate:
  - Split training data into training set and validation set (70/30 split is often used).
  - Train different models (e.g. diff. order polynomials) on training set.
  - Iterate through all possible hyper-parameters (e.g. polynomial M=1, then M=2, then M=3)
    1. Fix the hyperparameter (M=1 to start)
    2. Train a model using the training set
    3. Validate the model using the validation set
    4. Store the result for that hyperparameter
    5. Repeat from step 1 with the next hyperparameter.
  - Choose model (e.g. order of polynomial) with minimum error on validation set.
  - (not always done but often) Retrain the model on the complete training data (training set + validation set).

TEST ERROR/LOSS

- Now we need to be a little careful here.
  - Hyperparameters are parameters of the model that are not trained, i.e. we choose them.
  - As we can see though with hyper-parameter tuning, we are not exactly training them but we are not choosing them either.
  - The tuning that takes place is in some way using the validation set data.
  - It may not see the data directly, but it is getting a signal from it.
  - In effect, the validation data is compromised.
  - It doesn’t give us a true picture of the generalisation of the algorithm.
  - For this we still need a test set so that we can calculate a test error.

TEST SET

- You need to guard the test set carefully, from yourself, and use it only sparingly.
- The test set should be kept apart and almost never used.
- Choose your model hyperparameters using validation set
- Evaluate your final model with the test set.

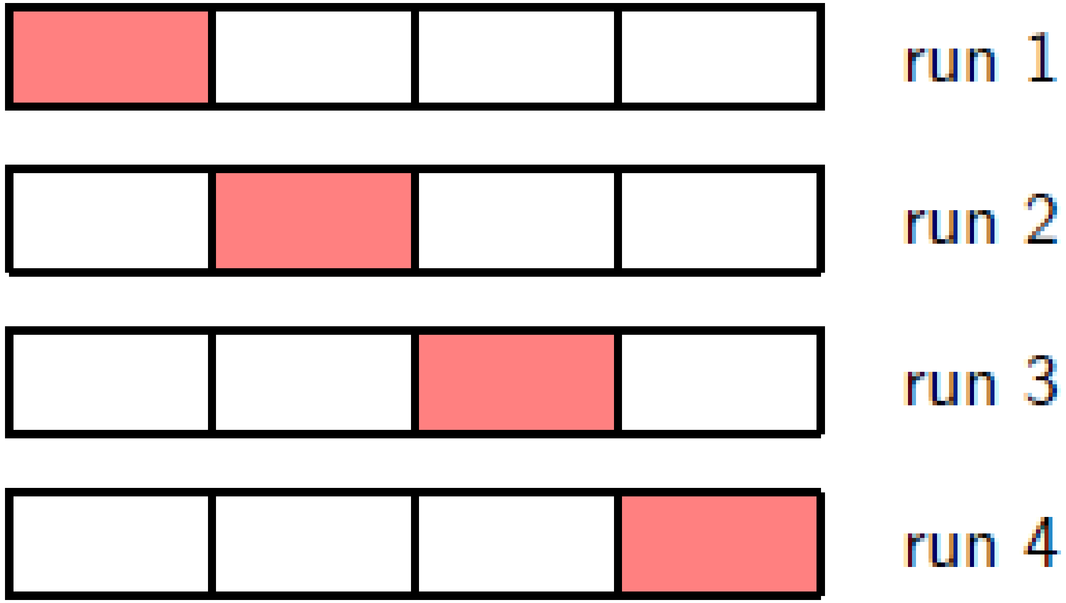
CROSS-VALIDATION

- Using a static Validation Set can introduce Bias. What if the validation set is not representative? If we have limited data this is more likely to occur.
- The idea of Cross-Validation is to do multiple runs at validation
  - Cross-validation creates S groups of data, use $S - 1$ to train, other to validate
  - Do a number of runs with these different groups (depending on Cross-Validation method chosen)
  - Average over all the scores (error) for the different training/validation splits
  - Choose the hyperparameter that has the lowest average error over all runs

CROSS-VALIDATION

- Cross-validation is an effective method for model selection, but can be slow
  - Models with multiple complexity parameters: exponential number of runs

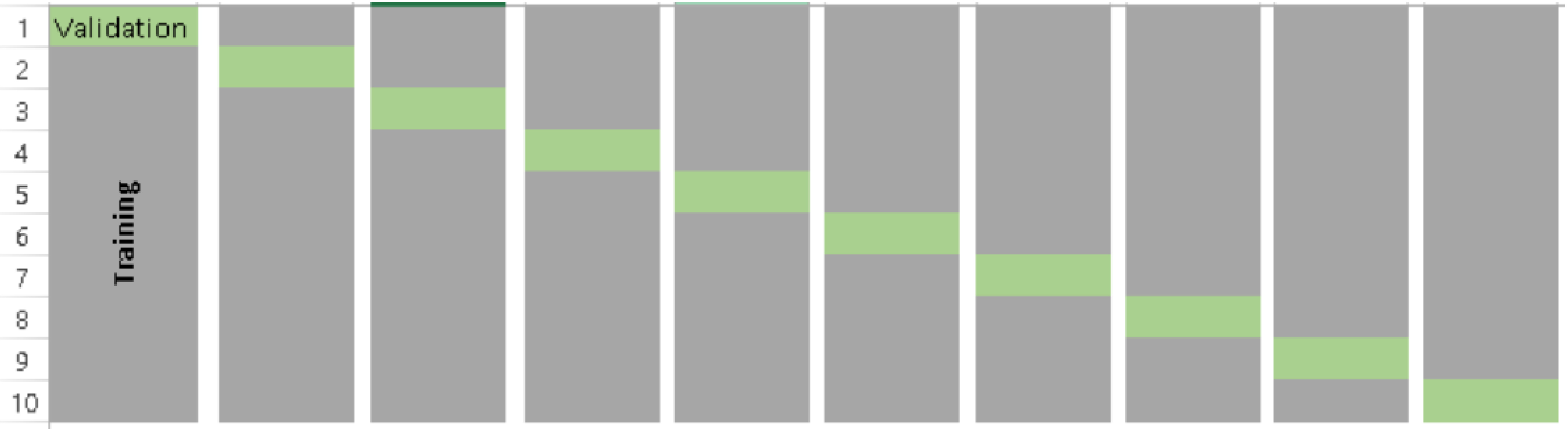
- Types of Cross-validation
  - Leave out one cross validation (LOOCV)
  - k-fold cross validation
  - Stratified k-fold cross validation
  - Adversarial validation
  - Cross validation for time series



K-FOLD CROSS VALIDATION

- Randomly split your entire dataset into k "folds"
- For each *k*-fold in your dataset, build/train your model on *k – 1* folds of the dataset. Then, evaluate the model to check the effectiveness for *k*th fold
- Record the error you see on each of the predictions
- Repeat this until each of the *k*-folds has served as the validation set
- The average of your *k* recorded errors is called the cross-validation error and will serve as your performance metric for the model
- (5 and 10 are usually often good choices for the number of folds - the default in sklearn is 5)

10-FOLD CROSS VALIDATION



FINAL PROCEDURE YET?

- Keeping the Test Data separate:
  - Iterate through all possible hyper-parameters (e.g. polynomial M=1, then M=2, then M=3)
    1. Fix the hyperparameter (M=1 to start)
    2. Randomly split the training data in k-folds
       - Train a model using $k - 1$ folds of the dataset.
       - Evaluate the model using the remaining fold.
       - Record the error.
       - Repeat until all folds have been a validation set.
    3. Take the average error of all folds and store that result for the particular hyperparameter
    4. Repeat from step 1, with the next hyperparameter
  - Choose model (e.g. order of polynomial) with minimum (average) error on the folds
  - (not always done but often) Retrain the model on the complete training data (training set + validation set).
- You choose your model using validation, you can evaluate your final model with the test set.

A PROCEDURE

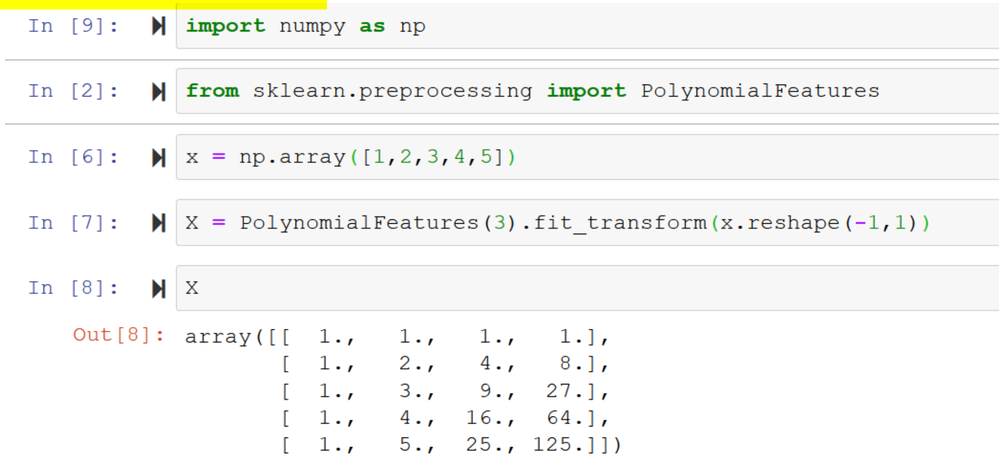
1. Split your labelled/known results into training set and test set
2. Select your base model (linear regression, nearest neighbours etc.)
3. Tune the hyper-parameters of the model using Cross Validation
4. Train the model (with chosen parameters) with the training set
5. Evaluate the model using the test set

POLYNOMIAL REGRESSION IN PYTHON

- Your dependent variable is y and independent variable x . You have a set of data X with a set of responses y. We want our model:

$$y = a_0x^0 + a_1x^1 + a_2x^2 + a_3x^3 + ...\, a_mx^m$$

- So we transform X to the polynomial degree we want.
- The weights a_0 etc. are still linear so it is still linear regression!



```
In [9]:  import numpy as np

In [2]:  from sklearn.preprocessing import PolynomialFeatures

In [6]:  x = np.array([1,2,3,4,5])

In [7]:  X = PolynomialFeatures(3).fit_transform(x.reshape(-1,1))

In [8]:  X

Out[8]:  array([[  1.,   1.,   1.,   1.],
                [  1.,   2.,   4.,   8.],
                [  1.,   3.,   9.,  27.],
                [  1.,   4.,  16.,  64.],
                [  1.,   5.,  25., 125.]])
```

- The lab-book on Moodle is to perform k-fold cross validation with Polynomial Regression to choose the degree of the polynomial.