`

3.1 ML Model Development Workflow

The prime objective of the ML modeling task is to obtain a model that generalizes well, which means that the model performs satisfactorily when supplied with data different from that used during model training. Achieving this takes more than just executing a 'model = <some ML_model>.fit()' command on the available data. In **C**hapter 1, we had a brief introduction to ML modeling workflow. In this chapter, we will learn about the different components of that workflow in more details. Figure 3.1 lists the common tasks you can expect yourself to carry out while building a ML model. While separate books can be written on each of these tasks, we will cover enough details on each of these to get you started on your ML journey.

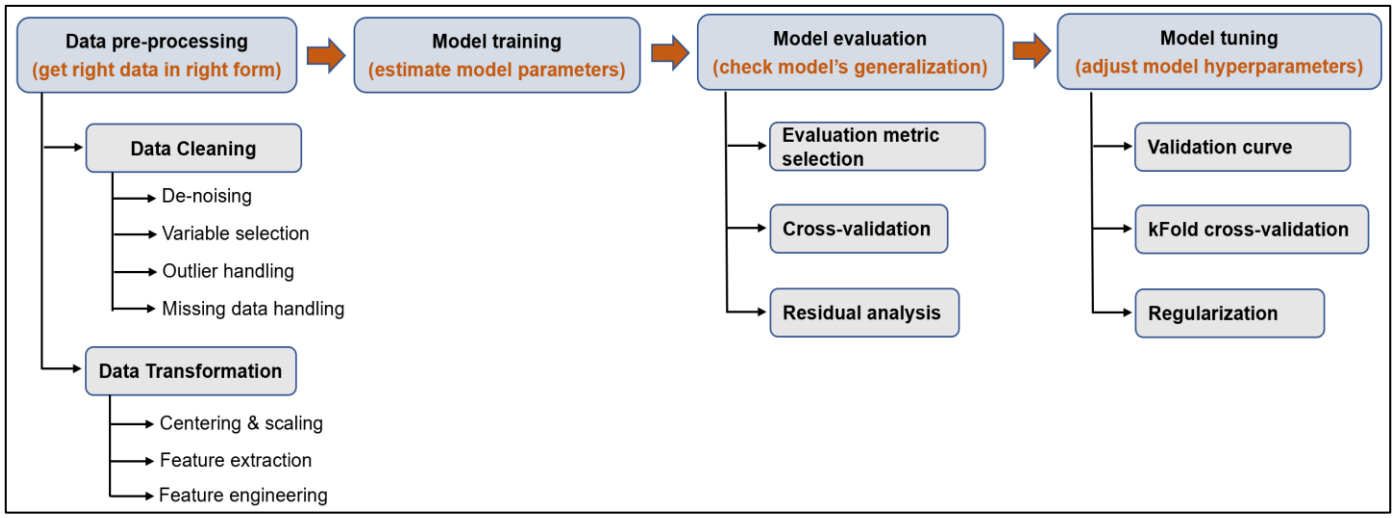


Figure 3.1: Recommended steps in a typical ML model development workflow

As you can see, there are four main steps in the workflow. In the first step, the available raw data are pre-processed to ensure 'correct' dataset is supplied to the subsequent model fitting step. This step can involve data cleaning (discarding irrelevant variables or corrupted/incomplete data samples) and data transformation (generating new features or modifying existing ones). While we will study data-transformation techniques in this chapter, data cleaning techniques are treated extensively in the next chapter. The next step of the workflow involves selection of a suitable model and estimating its parameters. While parameter estimation is handled by off-the-shelf model libraries available in Sklearn, model selection would require your judgement based on your expert knowledge of the specific process system. A practical guideline for model selection is Occam's razor which advices selection of the simplest model that generalizes well. After training, the model is evaluated to validate the modeling assumptions and assess model's generalization power. Model evaluation step often provides good hint about which part of the model (or hyperparameters) to tweak, which leads to the last step of model tuning. Specialized techniques are available to assist model tuning. With this broad overview of the workflow, let's now start learning about these techniques in detail.

`

3.2 Data Pre-processing: Data Transformation

Process data are seldom available in 'ready-to-model' format. The raw dataset may contain highly correlated variables or may have variables with very different value ranges. These data characteristics tend to negatively influence model training, and therefore, warrant data transformation to de-correlate or properly scale data. Sometimes, creating new features[11] out of provided variables can help to improve model accuracy. Overall, the aim of this pre-processing step is to transform given variables such that model training and model accuracy improves. Let's look at some commonly employed transformation techniques.

Data centering & scaling

In process dataset, it is common to find variables with very different order and spread of values. For example, you may have a temperature variable that varies between 1000°C to 1500°C, a concentration variable varying between 0.01M to 0.1M, and another temperature variable that goes from 300°C to 350°C. While the order and spread of these three variables are quite different, they may exert similar influence on the model output. However, unscaled dataset can result in one variable being given more importance than the other or slow model training. Centering and scaling transform variables to bring them on an equal basis! While some algorithms like PCA necessarily require centered data, as a best practice, you should always center and scale your dataset unless you have strong reason against it.

The most common technique is standard scaling or standardization, where each variable (in training dataset) is transformed to 0 mean and standard deviation of 1. Within an ML script, you will transform the test data using the computed mean and standard deviation of the training dataset. The illustration below shows how sklearn facilitates standardization

```
# import library, fit, and transform
import numpy as np
from sklearn.preprocessing import StandardScaler

X = np.array([[ 1000, 0.01,  300],
              [ 1200,  0.06,  350],
              [ 1500,  0.1, 320]])
scaler = StandardScaler().fit(X) # compute mean & std column-wise
X_scaled = scaler.transform(X)  # transform using computed mean and std
```

```
[[1.0e+03, 1.0e-02, 3.0e+02],
 [1.2e+03, 6.0e-02, 3.5e+02],
 [1.5e+03, 1.0e-01, 3.2e+02]]
```

```
[[-1.1355, -1.2675, -1.1355],
 [-0.1622,  0.0905,  1.2978],
 [ 1.2978,  1.177 , -0.1622]]
```

mean 0, variance = 1

[11] Feature and (independent) variable mean the same thing. While in ML literature, 'feature' is used to denote variables which go into predicting outputs, the term 'independent variable' is commonly used for this in statistics.

`

Using Sklearn library classes

In (object-oriented) computer programming parlance, in the above code, we created an object/instance (scaler) of StandardScaler class[12] and set its attributes, mean_ and var_, by calling the fit() method. They can be accessed as scaler.mean_ and scaler.var_. The fitted object (scaler) was then used to scale X by calling the object's transform method. This is the general pattern we will use with Sklearn's classes: objects will be initialized, properties fitted, and methods called.

Another popular technique is min-max scaling, also referred to as normalization. Here, each variable is transformed to a common range of values (typically 0 to 1). For 0 to 1 normalization, each variable is subtracted by its minimum value and then divided by its range. This is accomplished in Sklearn as follows

```
# import library, fit, and transform
import numpy as np
from sklearn.preprocessing import MinMaxScaler

X = np.array([[ 1000, 0.01,  300],
              [ 1200,  0.06,  350],
              [ 1500,  0.1, 320]])
scaler = MinMaxScaler () # create object
X_scaled = scaler.fit_transform(X)  # fit & transform
```

```
[[1.0e+03, 1.0e-02, 3.0e+02],
 [1.2e+03, 6.0e-02, 3.5e+02],
 [1.5e+03, 1.0e-01, 3.2e+02]]
```

```
[[0.  , 0.  , 0.  ],
 [0.4 , 0.56, 1.  ],
 [1.  , 1.  , 0.4 ]]
```

min 0, max = 1

Although standard scaling is often the default choice, there is no straight-forward answer to which scaling method is better. Either of the methods will usually work fine. Standard scaling is preferred if variables are Gaussian distributed or there are (a few) extreme outliers which can skew the 'min' and 'range' estimates during normalization.

Robust centering & scaling

If your dataset has a lot of outliers, then both the aforementioned methods will result in improper scaling. For example, in Figure 3.2, we can see that standard scaling isn't able to center the 'normal' data around 0 due to the presence of outliers. In such situations, robust MAD scaling should be employed, where, instead of mean and standard deviation, their robust counterparts, median and MAD (median absolute deviation from the median), are used. Mathematically, MAD scaling for any variable *x* takes the following form

[12] A class is like a blueprint defining the properties and behavior (functions) that are common among its objects. In OOP architecture, we can define a class once and then create its objects as many times as we want.

`

$$x_{MAD\ scaling} = \frac{x - median(x)}{MAD(x)}$$

Figure 3.2 shows that under MAD scaling, normal data does get centered around zero!

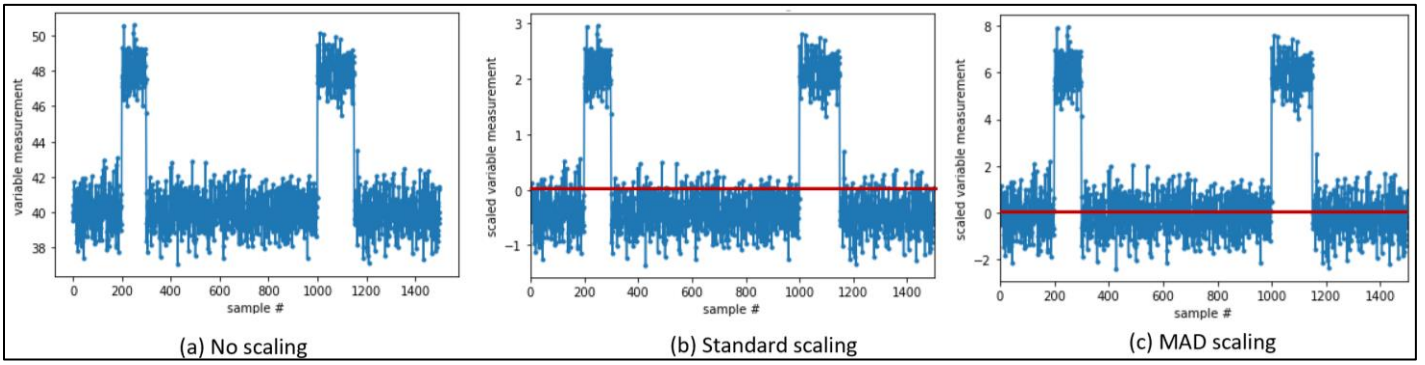


Figure 3.2: Application of standard and robust MAD scaling on data contaminated with outliers. Solid red line denoted the preferred centered location of 0. Complete code for generating these plots is available in online code repository.

Sklearn provides a RobustScaler class for robust scaling which is similar to MAD scaling, except that it uses interquartile range (IQR) instead of MAD as a robust estimator of data spread. Similar results can be expected with either of these robust scaling methods.

Feature extraction

Process systems have a lot of measurements which leads to increased data analysis complexity. However, you will often find that the original variable set can be reduced in dimensionality by extracting features which are combination of original variables and contain the same amount of information as the original variable set. This is a consequence of strong correlation among process variables.

Consider Figure 2 for example. Here, the three original variables (*x*, *y*, *z*) exhibit strong linear relationship to the extent that it is possible to transform (as shown) the 3D measurement space into a 2D (or even 1D) feature space while retaining most of the information about data variability. Accomplishing this in Sklearn is as simple as writing the following code

```
features = PCA(1).fit_transform(X_scaled) # extracting only 1 feature from scaled measurements
```

After feature extraction, rest of the analysis/model building is carried out using the extracted features. Specific techniques are available to decide optimal number of extractable features

`

for linear and nonlinear systems. Details on these techniques, concept of dimensionality reduction without 'losing information', PCA method, and other feature extraction methods such as ICA, FDA are covered in detail in Part 2 of the book. Apart from lower dimensionality, feature spaces may exhibit other nice characteristics such as uncorrelated or independent features, and, therefore, using these features instead of the original variables can generate better models.

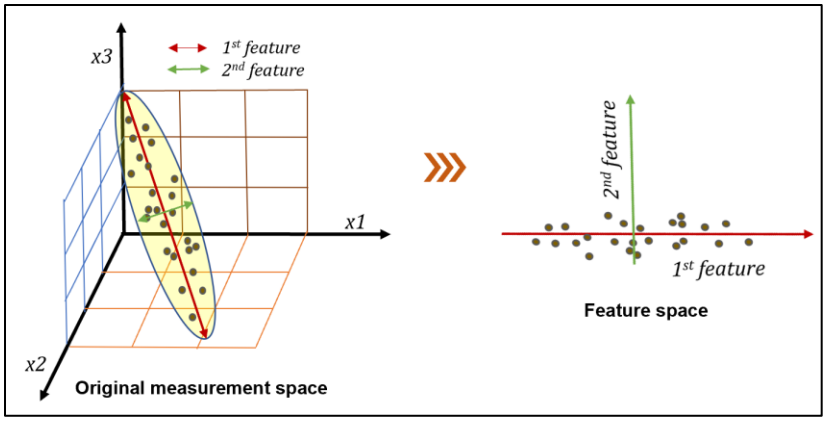


Figure 3.2: Extracting uncorrelated features from a 3D dataset

Feature engineering

While feature extraction is an unsupervised exercise (we do not explicitly specify the underlying correlation structure; algorithms extract relationships automatically), feature engineering involves careful and explicit transformation of variables which can assist ML algorithms in delivering better models. For example, if we examine the raw (*x* vs *y*) data in Figure 3.3, it is clear that a quadratic model may be needed. To accomplish this, we will engineer an additional feature (x^2) from the given *x* data and then fit a linear model between {*x*, x^2) and *y,* as shown in the code below

```
# read raw data
import numpy as np
data = np.loadtxt('quadratic_raw_data.csv', delimiter=',')
x = data[:,0,None]; y = data[:,1,None]

# generate quadratic features
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(x) # X_poly: 1st column is x, 2nd column is x^2

# scale model inputs
from sklearn.preprocessing import StandardScaler
```

`

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_poly)

# fit linear model & predict
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_poly, y)
y_predicted = model.predict(X_poly)
```

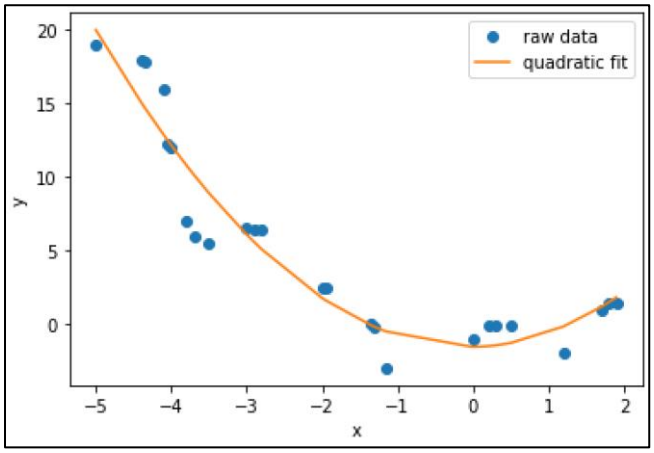


Figure 3.3: Linear model with quadratic feature

The advantage in the above example was that instead of doing a non-linear fit between *y* and *x*, we ended up doing just a simple linear fit. This is the power of feature engineering. It won't be an exaggeration to say that half of your task is done if you can engineer very relevant and informative features. Unfortunately, no well-defined generic approach currently exists for effective feature engineering. The task is system-specific and require domain knowledge. Nonetheless, let's look at a couple of feature engineering techniques that are commonly employed.

Mathematical transformations

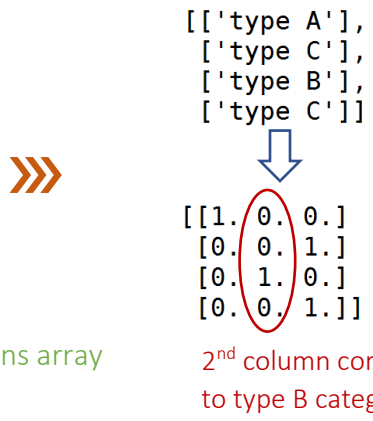
These transformations make use of functions like log, square, square root, product, temporal aggregation (using min, max, standard deviation), sin, tanh, etc. to create custom features. As seen in the previous illustration, these transformations allow usage of simple models, while making the effective relationship between the original variables more complex. Since, the selection of proper mathematical function is the most crucial and still largely a manual task, automated feature engineering (AFE) is currently an active research area in machine learning. Quite often, AFE techniques involve generating an exhaustive list of features from commonly used mathematical functions and then performing variable selection to select potentially useful features. However, universally accepted, generically applicable AFE methods don't exist yet and this makes your domain knowledge-guided feature engineering very important!

`

One-hot encoding

Most ML modeling algorithms require numerical data. But sometimes you may find yourself working with categorical data: for example, one of the model input variable may have entries like ‘plant type A’, ‘plant type B’, ‘plant type C’, etc. To handle such situations, you can convert the categorical data into numerical values. A naive approach would be to simply replace ‘type A’ with 0, ‘type B’ with 1, ‘type C’ with 2, and so on. However, this approach is erroneous unless there is a natural order among the plant type categories (maybe plant types A/B/C represent small/medium/large categories of plants). ML models may end up giving more importance to plant type C relative to types A (& B) due to type C’s higher designated value of 2. The preferred approach is to use one-hot encoding, which does the following transformation

```
# import class, fit, and transform
import numpy as np
from sklearn.preprocessing import OneHotEncoder

x = np.array([['type A'],
              ['type C'],
              ['type B'],
              ['type C']])
ohe = OneHotEncoder(sparse=False) # sparse=False returns array
X_encoded = ohe.fit_transform(x) # x in numerical form
```



As you can infer, a column has been generated for each unique plant type category (automatically identified by OneHotEncoder in the above code) and a binary value of 0/1 is assigned depending on occurrence of that category in a sample. For example, the 3rd sample is of ‘type B’ and therefore the 3rd entry in the 2nd column is assigned a value of 1.

Workflow automation via pipelines

In the previous example on linear modeling with quadratic features, we had to call fit and transform (or predict) methods multiple times for model training. These calls would be repeated for test dataset. For other models, you may have many more data transformers, such that it becomes slightly inconvenient to keep track of all transformations. To streamline this process, Sklearn provides a nice utility class called Pipeline, which allows us to chain together the transformers and estimators (predictive models) sequentially such that we need to call the fit method only once as shown in the code below for our quadratic model example.

`

```
# create pipeline for quadratic fit via linear model
# import relevant classes
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

# add transformers and estimators sequentially as list of tuples
# the names 'poly', 'scaler', 'model' can be used to access the individual elements of pipeline later
pipe = Pipeline([('poly', PolynomialFeatures(degree=2, include_bias=False)),
                 ('scaler', StandardScaler()),
                 ('model', LinearRegression())])

# fit pipeline and predict
pipe.fit(x, y)
y_predicted = pipe.predict(x)
```

The above code gives the same results as our previous code without pipeline and is more concise (and potentially easy to manage). Figure 3.4 shows the sequence of steps that got executed behind-the-scenes when we called the fit and predict methods of the pipeline. Really convenient, isn't it? Note that a pipeline does not have to include an estimator as its last element – you can only have transformers and call pipeline.transform() to transform your data through all the included transformers.

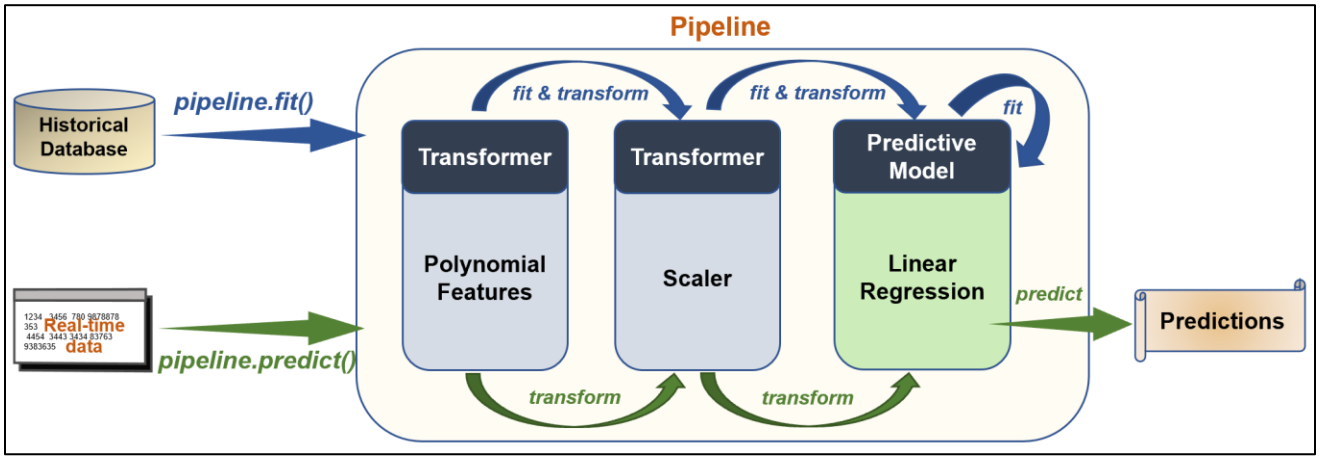


Figure 3.4: Streamlining ML workflow by combining transformers and predictors via pipeline

This completes our quick look into data transformation techniques. The takeaway message is that before dumping all the raw data for model training, you should always give some thought on what transformation you can apply to the raw data that can help the ML algorithm provide a good model. Carefully designed transformation can alleviate the need for large number of

`

training samples, allow usage of simple models, and reduce model training time. Next, we will see what techniques can be used to evaluate how well our model is performing.

3.3 Model Evaluation

After model parameters have been fitted, we need some means of assessing the model performance to see how well (or poorly) the model fits the training data. For this purpose, commonly computed metrics for regression and classification tasks are listed below.

Regression	Classification
• Mean Squared Error (MSE) • Mean Absolute Error (MAE) • Root Mean Squared Error (RMSE) • R Squared (R^2)	• Confusion Matrix • Accuracy • Precision, Recall, F1 Score • True positive rate (TPR), False positive rate (FPR)

Table 1: Common means of evaluating model performance

While it is easy to write our own python code to compute these metrics, it is even easier to use the sklearn.metrics module where these metrics are all implemented. We can import the desired metric class from the module and use it as shown below our quadratic fitting example.

```
# performance metric
from sklearn.metrics import r2_score
print('Fitting metric (R2) = ', r2_score(y, y_predicted))

>> Fitting metric (R2) =  0.91
```

Let us now take a quick look into the different metrics listed in Table 1.

Regression metrics

R-squared (R^2) is a good measure, technically for linear models, to determine how well the predicted output ($\hat{y}$) compares to the actual y. As seen in the metric formula table below, R^2 measures how much of variability in the output variable can be explained by the model. An R^2 of 1 indicates a perfect fit.

`

Regression Metric	Formula
MSE	$\frac{\sum(y-\hat{y})^2}{N}$
MAE	$\frac{\sum\lvert y-\hat{y}\rvert}{N}$
RMSE	$\sqrt{\frac{\sum(y-\hat{y})^2}{N}}$
R squared	$1-\frac{\sum(y-\hat{y})^2}{\sum(y-\bar{y})^2}$

Table 2: Formulae for common regression metrics. The summation is carried out over the *N* samples. $\bar{y}$ is mean of *y*

R^2 does not directly convey the magnitude of prediction errors. For this, the other metrics can be used. RMSE is the most popular metric for evaluating regression models and is computed as the square-root of the average of prediction errors. A related metric is the MSE which does not take the square-root. MAE is similar to MSE, except that it considers the absolute value of errors. Any of these metrics can be used to compare different models: smaller the values of these metrics (except R^2), better is the model fit.

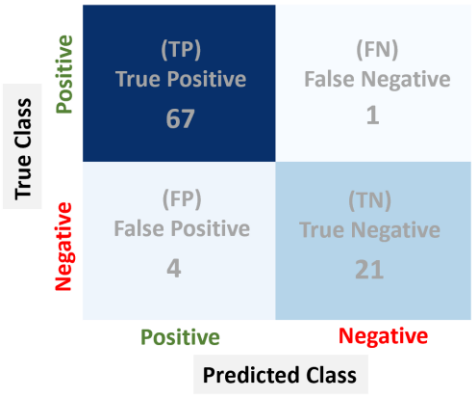
Several Sklearn models have their own score methods which computes an evaluation metric suitable for the problem they are designed to solve. For example, the `score()` *method of* `LinearRegression` *object returns the* R^2 *metric. Pipelines have a score method as well which simply executes the score method of the final model. For our quadratic fitting example, it can be used as follows*

```
# compute metric using score method
R2 = pipe.score(x, y) # no need to import metrics module
```

Classification metrics

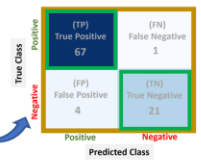
While in regression problems we care about how close the predicted values are to the actual values, in classification problems we care about the perfect match between the actual and predicted labels. Before we look at different classification metrics, let's first learn about a confusion matrix. The figure below shows a sample confusion matrix for a binary classification task; this matrix provides a comprehensive overview of how well a model has correctly (& incorrectly) classified samples belonging to the positive and negative classes. The shown matrix indicates that out of 25 (24 + 1) samples that actually belonged to negative class, the model correctly labeled 21 of those.

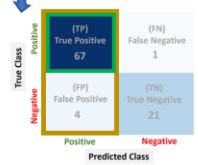
`

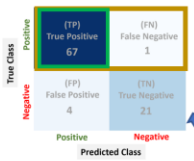


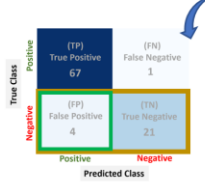
Confusion matrix can be used to compute several single-valued metrics, summarized in the metric formula table below. The most straightforward metric is 'accuracy' which simply communicates the ratio of true predictions to the total number of predictions. Accuracy can, however, be a misleading metric for imbalanced datasets (where more samples belong to one particular class). For example, in our example, overall accuracy is ~95% (88 out of 93), but we can see that the accuracy in correctly detecting negative class is only 84% (21 out of 25). Therefore, several other metrics are frequently employed to convey different aspects of model's performance.

Classification Metric	Formula
Accuracy	$\frac{TP+TN}{FP+FN+TP+TN}$
Precision	$\frac{TP}{TP+FP}$
F1 Score	$2\frac{Precision * Recall}{Precision + Recall}$









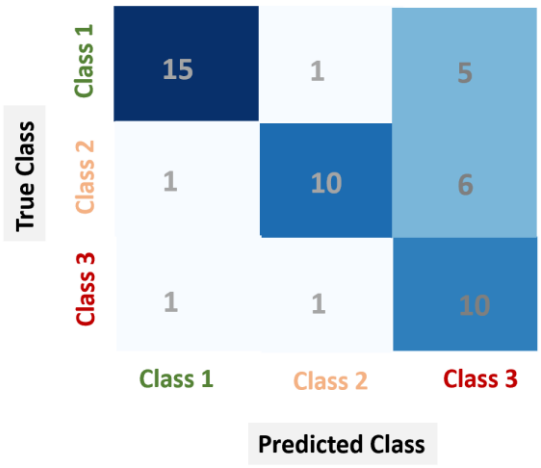
Classification Metric	Formula
Recall, TPR	$\frac{TP}{TP+FN}$
FPR	$\frac{FP}{FP+TN}$

Precision metric returns the ratio of number of correct positive predictions to the total number of positive predictions by the model while recall returns the ratio of number of correct positive predictions to the total number of positive samples in the data. For a process monitoring tool (where positive label implies presence of a process fault), recall denotes model's ability to detect process fault, while precision denotes accuracy of model's prediction of process fault. A model can have high recall if it detects process faults successfully, but occurrence of lots of false alarms will lower its precision. In other scenario, if the model can detect only a specific type of process fault and gives very few false alarms then it will have low recall and high precision. A perfect model will have high values (close to 1) for both precision and recall. However, we just saw that it is possible to have one of these two metrics high and the other low. Therefore, both metrics need to be reported. However, if only a single metric is desired then F1 score is utilized which returns a high value if both precision and recall are high and a

`

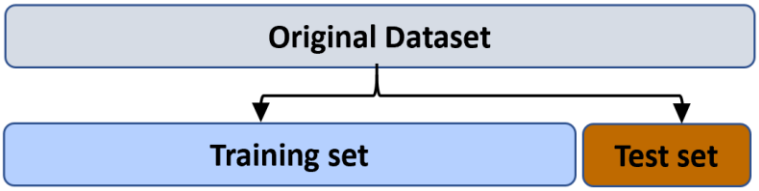
low value if either precision or recall is low. FPR (false positive rate) looks at the fraction of negative samples that get classified as positives. In our example, a high FPR of 0.16 slightly spoils the very rosy picture being presented by accuracy, precision, and recall numbers!!

For a multiclass problem, the confusion matrix would look something like the plot below where there is a row and column for each class type. Several important clues can be obtained from a multiclass confusion matrix plot. For example, in the plot below, it seems that some samples belonging to classes 1 & 2 tend to get classified as class 3 which suggest that some additional features should be added to help the model better distinguish between classes 1 & 2 and class 3. Metrics like precision can be computed by one vs all approach where you would choose one particular class as a positive class and the rest of the classes as a negative class.



Holdout method / Cross-validation

While the performance metrics computed using training data tell us how good our model fitting is, they are not true indicators of how well the model would perform for unseen data. For an unbiased generalization assessment, performance metrics should always be reported for a 'test' dataset that is not used during model training. This approach is called cross-validation. It's great if you already have a separate test dataset, otherwise the best practice is to split the available dataset into training and test sets as shown below.



`

Setting aside a portion of data for cross-validation is called holdout method. A good thumb-rule is to keep 20% of data in test set. Holdout can be accomplished in Sklearn using train_test_split utility function as shown below for our quadratic fit example

```
# separate test data
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1)

# check dataset sizes
print('Number of samples in training set: ', x_train.shape[0])
print('Number of samples in test set: ', x_test.shape[0])

>>> Number of samples in training set:  20
      Number of samples in test set:  5

# fit pipeline and predict
pipe.fit(x_train, y_train)
y_predicted_train = pipe.predict(x_train)
y_predicted_test = pipe.predict(x_test)

# performance metrics
from sklearn.metrics import mean_squared_error as mse

print('Training metric (mse) = ', mse(y_train, y_predicted_train))
print('Test metric (mse) = ', mse(y_test, y_predicted_test))

>>> Training metric (mse) =  3.79
     Test metric (mse) =  3.56
```

The random_state parameter in the train_test_split function is used to shuffle the dataset before applying the split. Shuffling dataset is a good practice to ensure that there is no unintentional data pattern between training and test datasets. For example, assume a dataset comes from a manufacturing plant and is sorted by production rates: without shuffling, we may end up with data only with low production rates in the training set, which is undesirable.

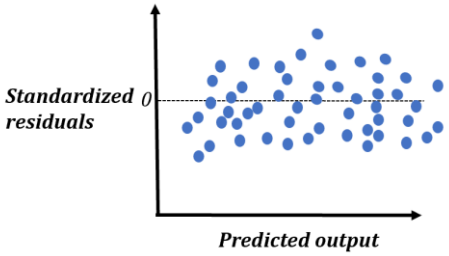
For classification problems, there is another important consideration to make. Ideally, we would like to ensure that the same class proportion that exists in the original dataset is retained in the training and test datasets. This approach is called stratified splitting and can be achieved through the stratify parameter in the train_test_split function. For example, we can pass a vector of fault / no-fault labels to this parameter and train_test_split function will ensure that the ratio of faulty and non-faulty samples is same in both training and test datasets.

`

After an estimate of model performance has been made using test data, the model may be re-trained, if desired, using the complete dataset. Since, an implicit assumption is always made that both the training and test datasets follow the same distribution, the model's generalization performance will only get better when re-trained with more data.

Residual analysis

Residual analysis is another way to validate the adequacy of (regression) models. It involves checking the model errors / residuals to ensure there are no undesired patterns in them. Presence of any obvious trend in residuals indicates a need for further data pre-processing or change of model. A convenient way to study the residuals is to simply draw scatterplots. Ideally, residuals would be small, centered around zero, independent of model inputs and outputs (predicted and measured), and uncorrelated in time (for dynamic models). An ideal residual plot would look something like this



In the above plot, no systematic pattern is apparent. If systematic patterns are present in residual plot then it implies that the model has not been able to extract all systematic pattern in the dataset. The specific patterns observed also often provide clues on potential remedies. Figure 3.5 shows some examples of 'bad' residual plots.

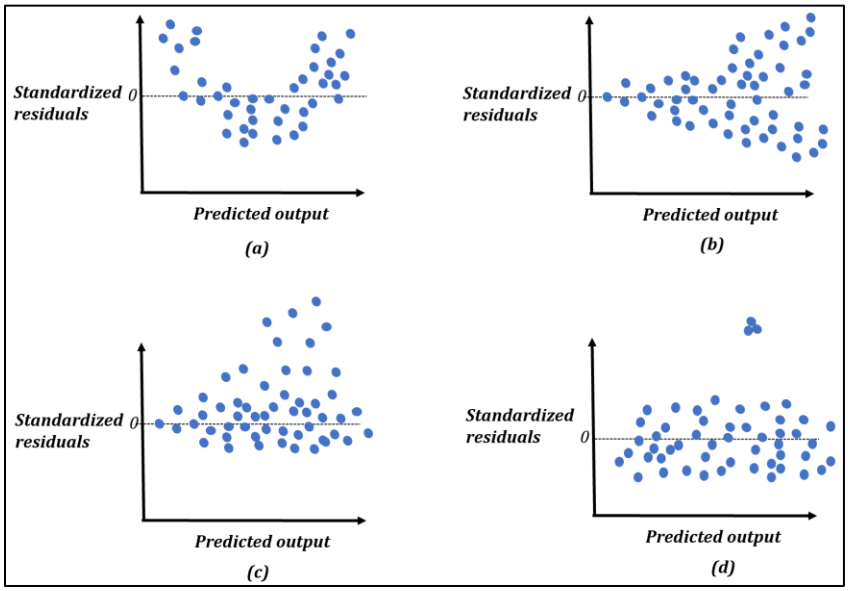


Figure 3.5: Examples of undesirable residual plots

`

In the above subplot (a), residuals show a nonlinear pattern which immediately conveys that the model is not able to adequately capture nonlinearities in the data. Nonlinear features may need to be included or a different model that can capture complex nonlinearities is warranted. In subplot (b) heteroscedasticity is apparent as residuals get larger for higher value of predicted output. Fixing this issue often entails appropriate transformation of variables such as log transformation. Heteroscedasticity may also indicate that a variable is missing. In subplot (c), while most of the samples show small residuals, several samples have high positive residuals. The suggested remedy in this case is the same as that for heteroscedasticity. Finally, in subplot (d), a group of samples appear as outliers. These outliers may need to be removed or further analyzed to check if these are normal samples; if samples turn out to be normal then model may need to be changed to account for the presence of these distinct groups of data.

For dynamic dataset, autocorrelation and cross-correlation (between residuals and past input/output data) function plots are used to check if the residuals are uncorrelated. Residual histograms or normal probability plots are also used to check if the residuals are normally distributed. If uncorrelatedness or normality assumptions are found violated, these plots, like scatter plots, give crucial hints on potential remedies.

We can see that residual analysis is a powerful diagnostic tool. As a best practice, you should always check the residual plots to ensure no modeling assumptions have been violated or no obvious room for improvement exists.

3.4 Model Tuning

Before we send our model for generalization assessment using test dataset, we need to ensure that the obtained model is the best model we can obtain. Or, if our model does not provide adequate generalization performance, some model adjustments are warranted. These tasks fall under the ‘model tuning’ step of the modeling workflow, and primarily involve optimizing values of model hyperparameters. *Model hyperparameters, unlike model parameters (which are the unknowns that are estimated during model fitting), are model configuration variables which need to be specified before model fitting*. For our quadratic fitting example, the weight coefficients of the linear model are the model parameters and the polynomial degree is the hyperparameter. For ANN models, number of neurons and layers are important hyperparameters, while weight coefficients are estimable model parameters. Finding optimal hyperparameter settings is also referred to as model selection in the ML literature.

`

Figure 3.6 shows the resulting model curves when our pipeline is fitted with different polynomial degrees. We can see that the model with high degree fits the training data better, but the fitting is a bit 'too much'! We would not be comfortable accepting this model although it has very low training error. How do we then select the optimal hyperparameter values before we make the final evaluation using test data? How can we diagnose if our model is fitting data too much or too less? What steps we can take to ensure that neither of these two undesirable situations occur? We will answer these important questions in the next few subsections.

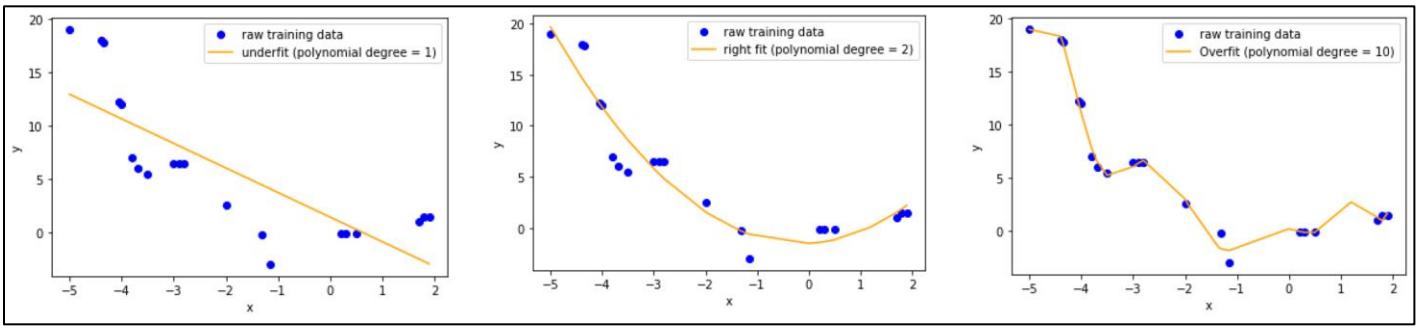


Figure 3.6: illustration of underfitting, 'just-right' fitting, and overfitting

Overfitting & underfitting

Underfitting occurs when our model is not able to capture the systematic relationships between predictors and response adequately (1st model in Figure 3.6) while overfitting occurs when the model captures even the non-systematic relationships such as noise (3rd model in Figure 3.6). As we increase model complexity by adjusting hyperparameter values, we move from underfitting end of the fitting spectrum to the overfitting end. A common way to visualize this is to plot validation curve using which we can also select our optimal hyperparameter value.

Overfitting is often the reason for poor performance of ML models. Overfitting is not necessarily only associated with 'complex models', and even simple linear regression models can overfit data (for example, if you have many irrelevant predictors or low number of training samples). Therefore, you should always pay careful attention that your model is not overfitting.

To plot validation curve, we will use a dataset that is kept separate from the data used for model fitting. We will call it validation dataset and the data used to fit the model as fitting or estimation dataset (you may wonder, 'what happened to the test dataset?' Hold onto that question, we will explain that shortly). Now, validation curve is simply a plot of model error (or any performance metric) for fitting and validation dataset for different hyperparameter values as shown in Figure 3.7. When hyperparameters are set such that model does not have enough

`

complexity or 'handles' to be able to describe the inherent structure in data, modeling errors will be large for both fitting and validation dataset leading to underfitting. As model complexity is increased, error on fitting dataset will decrease, however, the resulting model may not generalize well to the validation data leading to overfitting. Therefore, overfitting is characterized by small fitting errors and large validation errors. The sweet spot where both the fitting and validation errors are small, and validation errors just start increasing corresponds to the optimal hyperparameters.

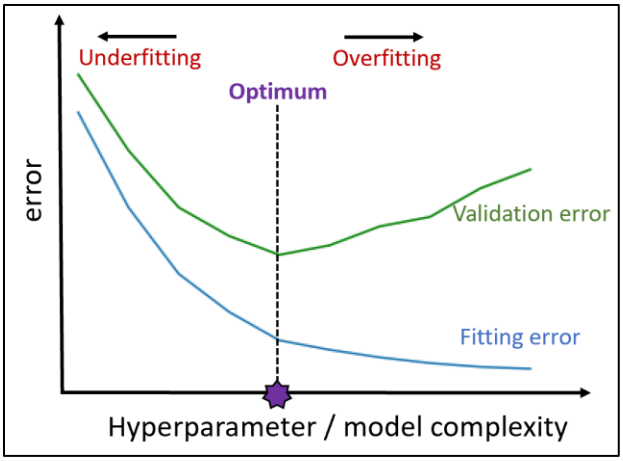


Figure 3.7: Typical validation curve

If your model is underfitting, your recourse should be to increase model complexity (for example by adding more depth in ANNs or using higher degree polynomials). You may explore adding more informative features. For overfitting, constrain model flexibility to prevent it from learning noise or random errors in fitting dataset. Removing irrelevant features or adding more fitting data can also help. If the optimal validation error is still very high, then a change of model may be needed (for example, using SVMs instead of decision trees).

The problems of underfitting and overfitting are also referred to as problems with high bias and high variance, respectively. High bias refers to the problem of model estimates being consistently far away from true values. On the other hand, if your model is overfitting, then your model estimates for the same validation dataset would vary considerably when different datasets are used for model fitting, leading to high variance in model estimates.

To generate the validation curve, you can use Sklearn's utility function validation_curve available in the model_selection module or write your own code as shown below.

```
# separate fitting and validation data
from sklearn.model_selection import train_test_split
```

`

```
x_fit, x_val, y_fit, y_val = train_test_split(x, y, test_size=0.2, random_state=1)

# assess performance on validation sets for different hyperparameter values
from sklearn.metrics import mean_squared_error as mse
fit_MSEs = []
validation_MSEs = []

for poly_degree in range(1,6):
    # set hyper-parameter value
    pipe['poly'].degree = poly_degree
    # fit & predict
    pipe.fit(x_fit, y_fit)
    y_pred_fit = pipe.predict(x_fit)
    y_pred_val = pipe.predict(x_val)
    # compute scores and append
    fit_MSE = mse(y_fit, y_pred_fit)
    validation_MSE = mse(y_val, y_pred_val)
    fit_MSEs.append(fit_MSE), validation_MSEs.append(validation_MSE)

# plot validation curve
from matplotlib import pyplot as plt
plt.figure()
plt.plot(np.arange(1,6), fit_MSEs, 'b--', label='fitting MSEs')
plt.plot(np.arange(1,6), validation_MSEs, 'g--', label='validation MSEs')
plt.legend(), plt.xlabel('Polynomial degree'), plt.ylabel('MSE')
```

Figure 3.8 shows the validation curve for our quadratic fitting example dataset. Although the distinction between fitting and validation MSEs are not very pronounced, 2nd degree seems to be the optimal hyperparameter value.

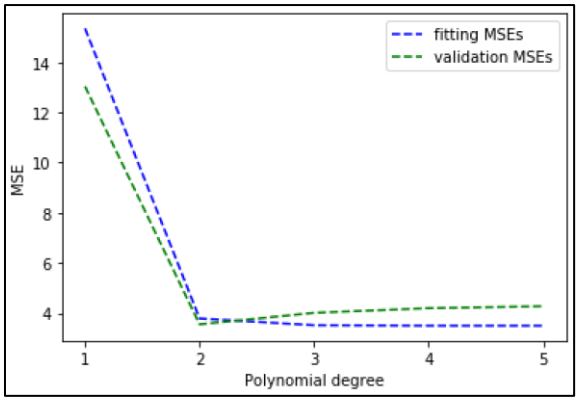


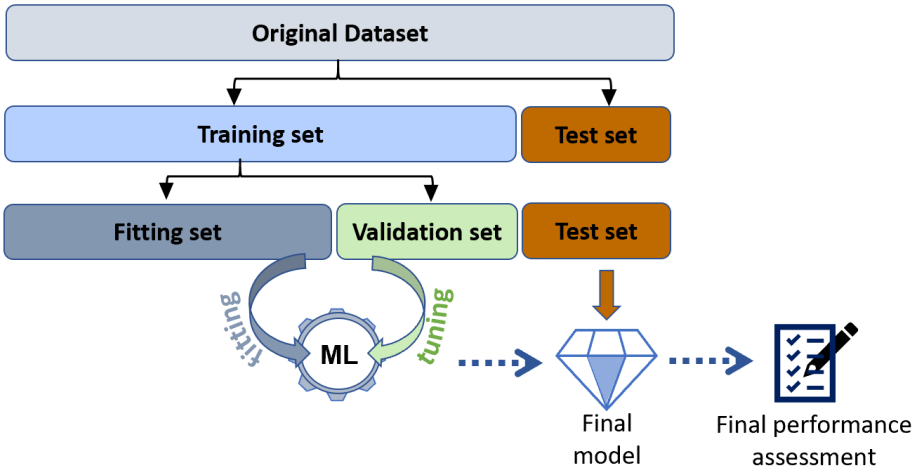
Figure 3.8: Validation curve for quadratic fitting example

`

A few more interesting things to note here. While validation MSEs are expected to be higher than fitting MSEs in general, here, we find them to be lower for the first 2 polynomial degrees. Also, if you extend this validation curve for even higher degrees, you will find validation MSEs decreasing instead of continuously increasing! So, does this mean that we can't trust inferences made using validation curves? You can actually trust these curves. What is happening here is that due to small dataset size, the validation dataset just happened to have data-points which show these unexpected behaviors. Infact, for small datasets, specialized multi-fold validation techniques exist which we will learn shortly.

Train/validation/test split

We used away the dataset that we had kept aside from the fitting dataset for model validation We can no longer use it as a test dataset for generalization assessment (since this dataset has already been 'seen' by the model and thus is no longer an unseen dataset). The solution is to set aside another chunk of dataset as shown below. This approach is called 3-way holdout method.



To implement 3-way holdout, we need to call train_test_split function twice as shown below

```
# train-validate-test split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1)
x_fit, x_val, y_fit, y_val = train_test_split(x_train, y_train, test_size=0.3, random_state=1)

print('Number of samples in fitting set: ', x_fit.shape[0])
print('Number of samples in validation set: ', x_val.shape[0])
print('Number of samples in test set: ', x_test.shape[0])

>>> Number of samples in fitting set:  14
Number of samples in validation set:  6
Number of samples in test set:  5
```

`

After model selection is completed, you may, if desired, retrain your model with the selected optimal hyperparameter settings on the combined fitting and validation dataset. Moreover, as mentioned before, after model performance assessment has been made using test data, the final model may be obtained by retraining on the entire dataset.

K-fold cross-validation

The 3-way holdout method is an elegant way of ensuring that model tuning and final assessment are based on unbiased estimates of model performance (i.e., using data not seen during model fitting and model training, respectively). However, there are several drawbacks as well with this method. If our original dataset is not large enough, we may not have the luxury of having 3 distinct subsets of data. Moreover, the performance estimates can be highly influenced by what data-points makeup the subsets and thus may lead to unfortunate selection of hyperparameters and final assessment.

The popular alternate solution is *k*-fold cross validation. The idea is simple. Instead of having a single test (or validation) subset, multiple subsets are created, and overall performance estimate is computed as the average of performances on individual subsets. Figure 3.9 illustrates this idea where *k*-fold cross validation is used for both final performance assessment and model tuning.

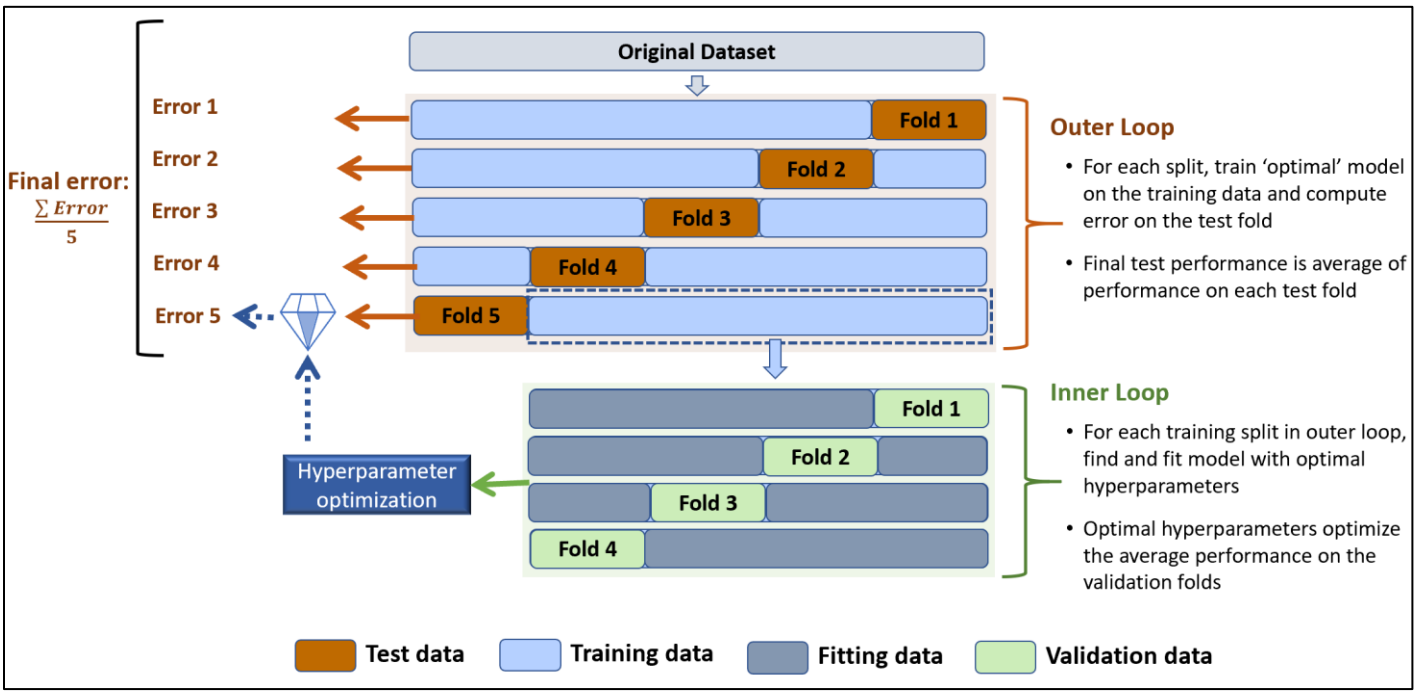


Figure 3.9: Nested *k*-fold cross validation for model tuning (inner loop) and final performance assessment (outer loop)

`

To understand Figure 3.9, let's consider the outer loop first

- 5-fold cross validation is employed for final performance assessment
- The original dataset is split into 5 folds and 5 separate models are trained
- For each model, one of the folds is used as test data and the rest of the data are used for model training
- The overall model performance is simply the average performance of the 5 models

For model training in the inner loop

- 4-fold cross validation is employed, i.e., the training data is split into 4 folds
- For each hyperparameter set, 4 separate models are fitted using the fitting datasets
- The overall validation performance is simply the average performance of the 4 models on their respective validation folds
- The hyperparameter values that optimize the overall validation performance is selected and the model is re-trained using the complete training data before being sent to the outer loop.

Very often, *k*-fold cross validation is only employed for model tuning and the final performance assessment is made using only a single test holdout set. This conventional cross-validation approach is illustrated in Figure 3.10.

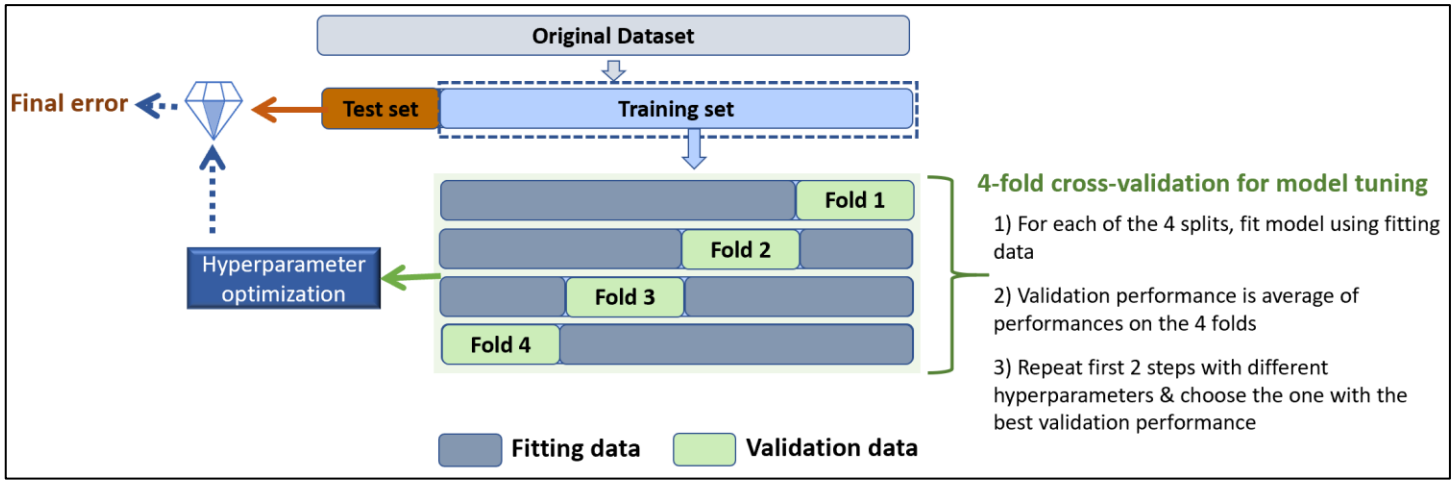


Figure 3.10: *K*-fold cross validation for model tuning

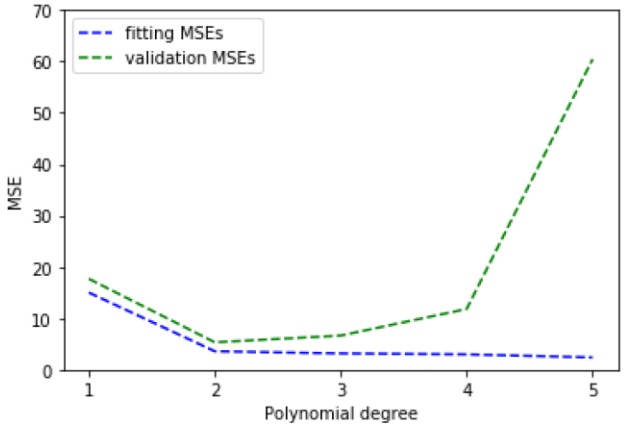
Let's apply *k*-fold cross validation technique to find the optimal polynomial degree with our illustration dataset. We will use the KFold utility function provided by Sklearn which returns the indices of data-points that belong to the fitting & validation subsets in each of the *k* splits.

`

```
# separate training and test data
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1)

# generate k=3 folds on training data
from sklearn.model_selection import KFold
kfold = KFold(n_splits = 3, shuffle = True, random_state = 1)

# compute validation performances using the K folds
from sklearn.metrics import mean_squared_error as mse
overall_fit_MSEs = [] # overall errors for different polynomial degrees
overall_val_MSEs = []

max_polyDegree = 5
for poly_degree in range(1,max_polyDegree+1): # loop over hyperparameters
    pipe['poly'].degree = poly_degree

    split_fit_MSEs = [] # errors for different splits
    split_val_MSEs = []

    for fit_indices, val_indices in kfold.split(x_train): # loop over splits
        x_fit = x_train[fit_indices]
        y_fit = y_train[fit_indices]
        x_val = x_train[val_indices]
        y_val = y_train[val_indices]

        # fit & predict
        pipe.fit(x_fit, y_fit)
        y_pred_fit = pipe.predict(x_fit)
        y_pred_val = pipe.predict(x_val)

        # compute average error and append
        split_fit_MSEs.append(mse(y_fit, y_pred_fit))
        split_val_MSEs.append(mse(y_val, y_pred_val))

    overall_fit_MSEs.append(np.mean(split_fit_MSEs))
    overall_val_MSEs.append(np.mean(split_val_MSEs))
```

Figure below shows the resulting validation curve which is more along the expected lines. Note that Sklearn provides a cross_val_score function which can be used to replace the loop over the splits in our code above.

`



K-fold cross-validation is the default cross validation techniques inbuilt in several of Sklearn's functions such as validation_curve. The downside to this technique is high computational cost as model fitting is carried out multiple times. Therefore, for deep learning-based solutions, which anyway employs large datasets, 3-way holdout method is often the preferred choice.

Choosing number of folds

The typical (and often the default) value of *k* in *k*-fold cross-validation is 10. For large datasets, smaller value (*k* = 3 or 5) can be used. When we increase *k*, less samples are put in the validation set; this results in unpreferable higher variance in the validation scores among the different splits. An extreme scenario is when number of folds equals the number of training samples. This is referred to as leave-one-out cross validation and is employed when number of training samples is very limited.

Regularization

If your model is overfitting, one of the things to do is to check the values of the model parameters (weight coefficients for linear models). You may find some or all of them with very large values. This is problematic because the model becomes very sensitive to input variables with high coefficients, i.e., model predicts big response changes for even very small variations in these input variables. This is unlikely to be true for real process systems. For example, following are the model coefficients with 10-degree polynomial features for our illustration dataset. Look at the higher degree coefficients

```
# get model coefficients
print(pipe['model'].coef_) # does not include intercept term

>>> [-1.96123e+00 -1.70142e+01  1.61112e+02  5.53503e+02  -1.47185e+03 -5.24500e+03
9.88824e+02  1.58830e+04  1.72273e+04  5.73349e+03]
```

`

Bayesian Information Criterion (BIC) for model selection

If you want to utilize all your training data for model fitting or validation subset cannot be set aside for model selection (e.g., when using Gaussian mixture models), BIC technique can be used which provides an optimal trade-off between model complexity (or number of model parameters) and model accuracy. We have seen that model accuracy on training data can be increased by adding more model parameters, but this can also lead to overfitting. BIC resolves this conundrum by minimizing a metric that includes a penalty term for the number of model parameters.

Mathematically, $BIC = n_p ln(n) - 2ln(\hat{L})$, where

n_p equals number of estimated parameters
n equals number of data-points or observations
$\hat{L}$ equals the optimal value of likelihood function (model accuracy)

We can see that a model with high accuracy/likelihood and fewer parameters will result in low BIC value, and hence, optimal model corresponds to the minimum BIC value. BIC owes its popularity for its effectiveness and low computational cost: It has been shown theoretically that for sufficiently large training data, if the correct model is included among the set of candidate models, then BIC is guaranteed to select it as the best model. We will see an application in Chapter 8.

Another popular model selection metric is AIC (Akaike information criterion). The penalty term is smaller in AIC. Consequently, AIC tends to select more complex models compared to those selected by BIC. Do remember that more complex models might overfit while simpler models might underfit. Therefore, selection between AIC and BIC methods is problem dependent.

Therefore, to keep model parameters 'simple', modeling algorithms are tweaked to decrease the likelihood of overfitting. This modulation is called regularization and is a widely used approach to tackle overfitting. You should always employ some form of regularization in your model.

Common mechanisms of regularization

In one of the forms of regularized model, the objective function (which is minimized during fitting) includes an additional term that is proportional to summation of square of model coefficients ($\boldsymbol{\theta}$) as shown below

`

$$J_{regularized}(\boldsymbol{\theta}) = MSE(\boldsymbol{\theta}) + \alpha \sum_i \theta_i^2$$

The additional term penalizes the algorithm for choosing high coefficient values. Therefore, during model fitting, the algorithm will be forced to minimize fitting error while keeping coefficients small. This version of regularization is called L2 or ridge regularization. The 'α' term is a hyperparameter which modulates the magnitude of the penalty term.

In another form of regularization, the objective function is modified as follows

$$J_{regularized}(\boldsymbol{\theta}) = MSE(\boldsymbol{\theta}) + \alpha \sum_i |\theta_i|$$

Here, the penalty term is simply proportional to the summation of absolute value of the coefficients. This is called L1 or lasso regularization. Let's see how these mechanisms help us with the overfitting problem with our 10th degree polynomial fitting. L1 / L2 can be implemented by just switching the linear model in the pipeline with the regularized linear models as shown below

Ridge regression

```
from sklearn.linear_model import Ridge

pipe_L2 = Pipeline([('poly', PolynomialFeatures(degree=10,include_bias=False)),
                    ('scaler', StandardScaler()),
                    ('model', Ridge(alpha=0.1))])
```

Lasso regression

```
from sklearn.linear_model import Lasso

pipe_L1 = Pipeline([('poly', PolynomialFeatures(degree=10,include_bias=False)),
                    ('scaler', StandardScaler()),
                    ('model', Lasso(alpha=0.1))])
```

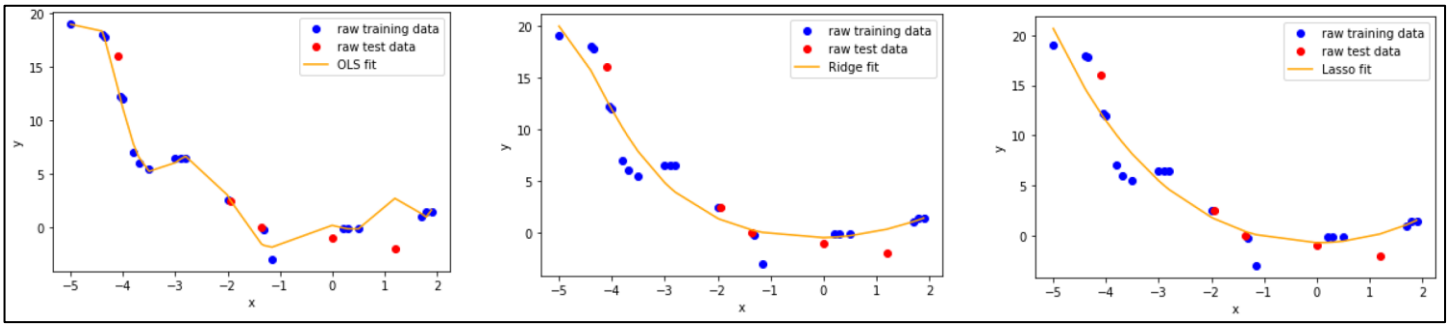


Figure 3.11: Non-regularized and regularized polynomial fitting

Figure 3.11 confirms that regularization works!! To see what is happening under the hood, let's check the model coefficients of the regularized models

```
# get model coefficients
print(pipe_L2['model'].coef_)
```

```
>>> [ 0.27326  3.49270 -0.35861  0.59074 -1.20522  1.90652 -1.80640  0.86690  0.87244
-3.20685]
```

`

```
print(pipe_L1['model'].coef_)

>>> [-0. 4.38109181 -0.09273215  1.679048   -0.  0.  -0.  0.  0. -0. ]
```

While both L1 and L2 regularizations ensure that model coefficients remain small, Lasso actually has pushed several coefficients to zero. This is representative of the general behavior of Lasso which is often used for variable selection as the coefficients of irrelevant variables get set to zero. The advantage of L2 is that it is computationally favorable as the penalty term is differentiable unlike L1's penalty term. The choice between L1 and L2 is problem dependent and requires trial-and-error.

L1 and L2 regularizations can be combined into elastic-net regularization. Not all forms of regularizations involve putting penalty terms in objective function. You will study methods like dropout, batch normalization, early stopping in Chapter 11, which help to ensure model coefficients remain 'stable' to avoid overfitting.

You will have noticed that we have talked about regularizing only the weight coefficients and not the intercepts. This is true in general. Large intercepts (or biases in neural networks) will not cause large response changes upon small variations in inputs, and therefore, regularizing them is often not needed.

Choosing value of α

The 'α' term is an important hyperparameter which should be carefully specified before model fitting. Very small α will make regularization ineffective potentially leading to overfitting. Very high value will push the model coefficients to zero causing underfitting. The optimal value is difficult to know before-hand and need to found via cross-validation. However, a general recommendation is to keep α low rather than high.

Hyperparameter optimization via GridSearchCV

Previously we used validation curves to find the optimal hyperparameters. Sklearn provides a GridSearchCV utility function that we can leverage to do this task more conveniently. GridSearchCV takes lists of potential hyperparameter values and does an exhaustive search to find the optimal combination of hyperparameter values. The code below shows the implementation.

```
# separate training and test data
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1)
# find optimal hyperparameter via GridSearchCV
```

`

```
from sklearn.model_selection import GridSearchCV

param_grid = {'poly__degree': np.arange(1,6)}
gs = GridSearchCV(pipe, param_grid, scoring='neg_mean_squared_error', cv=3)
gs.fit(x_train, y_train)

print('Optimal hyperparameter:', gs.best_params_)

>>> Optimal hyperparameter: {'poly__degree': 2}
```

As expected, GridSearchCV also returns the optimal polynomial degree as 2. Note that we are using 3-fold cross validation here as specified via cv=3. By default, GridSearchCV refits the estimator on the complete supplied dataset using the best found hyperparameters. The best estimator can be obtained via gs.best_estimator_ for subsequent predictions (predictions can also be done via gs.predict function). In the above code we explicitly supplied mean squared error as the scoring metric; the default behavior is to use the estimator's default scorer which would be R-squared for linear regression in Sklearn.

This concludes our quick look into the standard ML model tuning approaches and ML modeling workflow. You now know how raw data should be transformed before model fitting, how to evaluate and report model's performance, and how to find optimal model hyperparameters during model tuning.

Summary

In this chapter, we learnt the best practices of setting up a ML modeling workflow. During the course of learning the workflow, we saw several nice utilities (pipelines, *k*-fold, grid-search, etc.) provided in Sklearn library which make ML scripting easier and manageable. The takeaway message from this chapter should be that ML modeling is a collaborative work between you and machine. This chapter exposed you to your side of the task!!