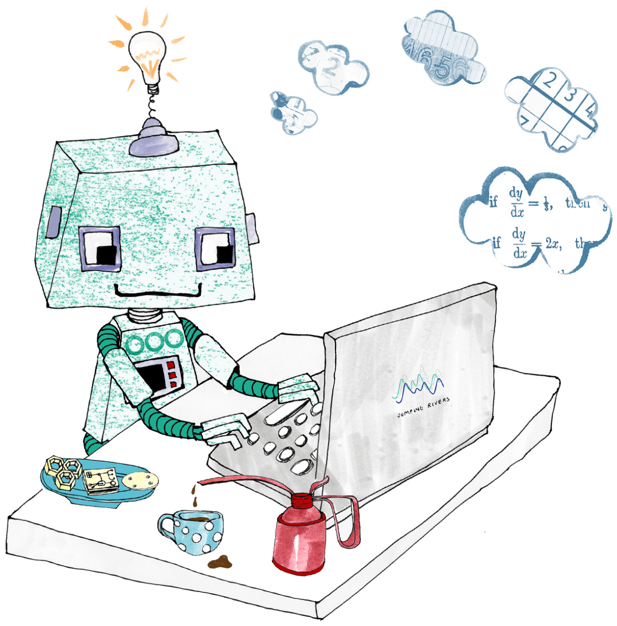
MACHINE LEARNING IN PYTHON

jumping rivers

jumpingrivers.com

CREATING CLARITY WITH YOUR DATA.

Saving organisations like yours thousands of work hours.



Jumping Rivers is an analytics company whose passion is data and machine learning. We help our clients move from data storage to data insights. Our trainers and consultants come with over 40 years combined experience in R, Python, Stan, Scala and other programming languages.

Consultancy

- RStudio certified - one of only seven full-service partners.
- We have plans for managing all RStudio products. From on-demand support to full care packages.
- As an RStudio reseller, we can provide free consultancy.
- Experts in both data science and data engineering.
- Creation of bespoke dashboards and Shiny apps.
- Optimisation of your algorithms.

Training

- R/Python/Stan/Scala
- The Tidyverse
- Machine Learning and Tensorflow
- Dashboards with Shiny
- Statistical Modelling
- And many more!

jumping rivers
jumpingrivers.com
info@jumpingrivers.com
@jumping_uk

1 Introduction

To begin, a couple of quotes from Wikipedia.

1.1 What is analytics?

> Analytics is the discovery, interpretation, and communication of meaningful patterns in data and applying those patterns towards effective decision making.

1.2 What is machine learning?

> Machine learning (ML) is the study of algorithms and statistical models that computer systems use to progressively improve their performance on a specific task. Machine learning algorithms build a mathematical model of sample data, known as “training data”, in order to make predictions or decisions without being explicitly programmed to perform the task. ... In its application across business problems, machine learning is also referred to as predictive analytics.

The corresponding entry for predictive analytics includes

> Predictive analytics encompasses a variety of statistical techniques from data mining, predictive modelling, and machine learning, that analyse current and historical facts to make predictions about future or otherwise unknown events.

I'M JUST OUTSIDE TOWN, SO I SHOULD BE THERE IN FIFTEEN MINUTES.
ACTUALLY, IT'S LOOKING MORE LIKE SIX DAYS.
NO, WAIT, THIRTY SECONDS.
THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

Figure 1.1: https://xkcd.com/612/

1.3 A statistical model

Suppose we observe some response, Y, that we are interested in together with a set of p predictors, $X = (X_1, X_2, \ldots, X_p)$, and that we believe there to be some relationship between Y and X. In general we could express this as

$$Y = f(X) + \epsilon$$

where $f(\cdot)$ is some fixed but unknown, essentially arbitrary, function of the predictors X and ϵ is a random error independent of X. Here $f(\cdot)$ represents the systematic information about Y provided by X. We call this a statistical model as it describes the relationship between one or more random variables, the quantities are not deterministically related.[1] Knowledge of the true $f(\cdot)$ is almost certainly unavailable but given a set of observations we can estimate it. In analytics and machine learning we are describing the process of estimating the function $f(\cdot)$ using a set of observed data.

[1] i.e being taller doesn't always mean you have bigger feet.

2

Simple Regression Techniques

2.1 Linear regression

Linear regression is a simple[1] approach to statistical learning and a useful tool for predicting a quantitative response. Though it is relatively simple when compared to some of the more advanced techniques it is still a widely used and useful technique.

[1] and computationally fast, so it's worth remembering.

It also provides an ample starting point to learning about analytics an machine learning as many techniques can be considered as extensions to linear regression. In addition we can start to build up our machine learning workflow in a context where model complexity is not a burden.

Simple linear regression

Simple linear regression is the most straightforward approach for predicting a quantitative response Y on the basis of a single predictor X. We make the assumption that there is a linear relationship between Y and X, in particular we specify our model as having the form

$$Y = \beta_0 + \beta_1 X$$

where β_0 and β_1 are the slope and y-intercept respectively.

Estimating the coefficients

In practice, β_0 and β_1 are unknown. To make predictions or inferences from our model we first need to find the straight line that gets as close as possible to all of the data points, i.e.. we need to find the best choice of β_0 and β_1 given a set of data.

2.2 Example: Boston housing data

Fitting a simple linear regression model in Python is easy using the fantastic **sklearn** package.[2]

[2] **sklearn** contains implementations to a whole host of model techniques together with other tooling for a standard applied analytics workflow.

Load the modules that we need to get started.

```
from sklearn.datasets import load_boston
from sklearn import linear_model
```

The `load_boston()` function provides the covariates and response as separate attributes in the resultant object. So we can assign both the `X`, input data, and `y`, response target, using[3]

```
boston = load_boston()
X, y = boston.data, boston.target
```

[3] There is more information about the study and the variables in the `boston` object. Try `dir(boston)` to see all the attributes.

Visualising the data

Personally I find it easier to work with **pandas** `DataFrames` to visualise data as we can make use of the excellent **seaborn** package and it's convenient syntax with `DataFrame` objects. We can turn our current `X` and `y`, which are both **numpy** `arrays` into a single `DataFrame`, with

```
import pandas as pd
df = pd.DataFrame(X, columns = boston.feature_names)
df['MEDV'] = y
```

We can produce a simple scatter plot of the response `MEDV` against the first predictor `CRIM`

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.scatterplot(x = 'CRIM', y = 'MEDV', data = df)
plt.show()
```

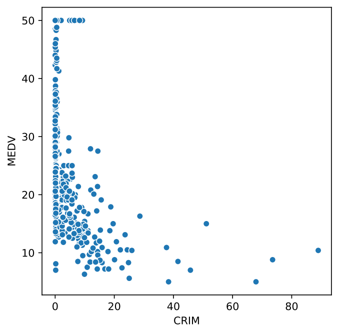


Figure 2.1: A simple scatter plot of MEDV vs CRIM.

Model fitting

To fit a model using just a single predictor we first extract the training variables.

```
X_train = df['CRIM']
y_train = y
```

Unfortunately, **sklearn**'s various model fitting functions typically expect a two dimensional array for the covariates. Since we have extracted only a single feature here it is only one dimensional. We need to reshape the `X_train` values to be the appropriate shape.[4]

```
if len(X_train.values.shape) == 1:
  X_train = X_train.values.reshape(-1, 1)
```

[4] This is not necessary if using more than a single feature.

Create a `LinearRegression` object.[5]

```
model = linear_model.LinearRegression()
```

[5] This object is of a broader class of estimator objects.

We then pass the data to the model object's `.fit()` method.

```
model.fit(X_train, y_train)
```

We can make predictions from our fitted model with the `.predict()` method.

```
import numpy as np
new_value = np.array(1, ndmin = 2)
model.predict(new_value)
#> array([23.6179159])
```

```
multiple_values = np.array([1, 2, 3], ndmin = 2).T
# here we use .T to make it a column vector
model.predict(multiple_values)
#> array([23.6179159 , 23.20272562, 22.78753534])
```

Fitted values

Fitted values of a model typically describes the predicted $\hat{y}$ for the observations X. To get the model fitted values we could just predict from the model using the values used to train it.

```
fitted = model.predict(X_train)
```

And then create a plot with the added fitted line.

```
ax = sns.scatterplot(x = 'CRIM', y = 'MEDV', data = df)
sns.lineplot(df['CRIM'], fitted, ax = ax)
plt.show()
```

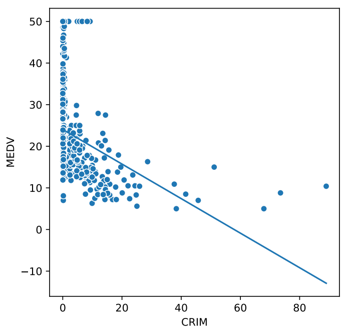


Figure 2.2: Scatter plot with fitted line

Interpreting the coefficients

The coefficients of the fitted model are kept in the `model.coef_` attribute.

```
model.coef_
#> array([-0.41519028])
```

This gives us the expected change in y for a unit change in X.

2.3 Multiple linear regression

In practice we typically have more than a single predictor. The extension to multiple linear regression is trivial. If we have p predictors, the model has the following structure

$$Y = f(X) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$$

That is we add a gradient coefficient for each of the predictors whose information we want to include about Y. We will now use the first few variables from Boston housing data. Again we should do some exploratory graphical analysis of the features.

```
X_train = df.iloc[:,:3]
grid = sns.PairGrid(data=pd.concat([X_train,pd.Series(y_train,name="MEDV")],axis = 1))
grid.map_offdiag(sns.scatterplot)
grid.map_diag(sns.distplot)
plt.show()
```

If we are happy, we can proceed to fit the model in the same way.

```
model.fit(X_train, y_train)
```

And then make predictions

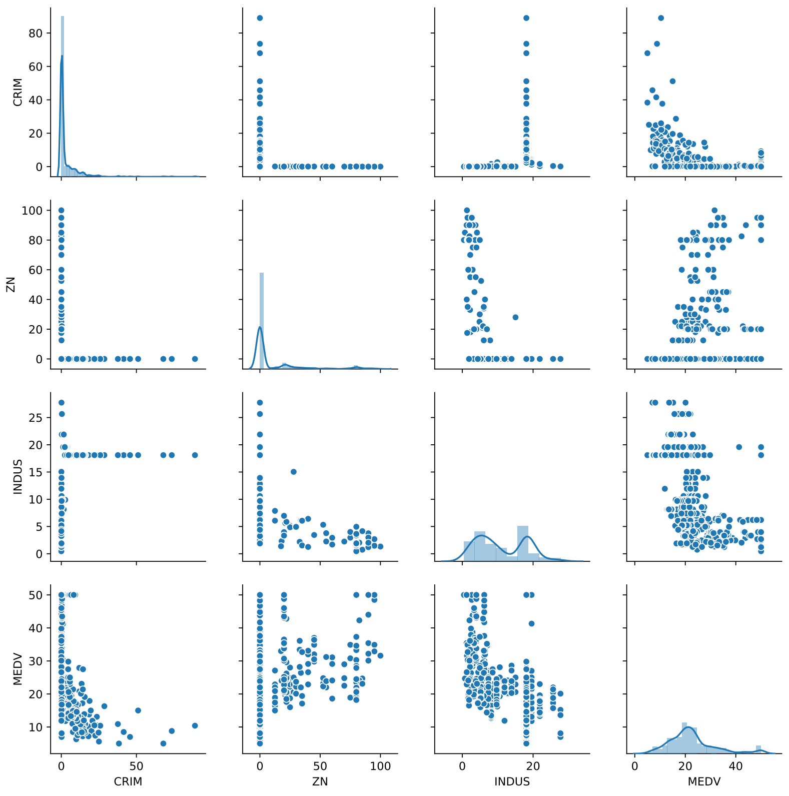


Figure 2.3: Exploring relationships between predictors and outcome for the boston housing data.

```python
new_values = np.array(X_train.mean(), ndmin = 2)
model.predict(new_values)
#> array([22.53280632])
```

Residuals

Given model predictions $\hat{y}_i$ for each observation X_i we can define a residual as

$$\epsilon_i = y_i - \hat{y}_i$$

Residuals effectively show us what our model has failed to capture[6]. Another useful term here is

$$RSS = \sum_{i=1}^{n} \epsilon_i^2$$

the sum of squared residuals. This gives us an overall score of how close to the observations the model is getting.[7]

Analysis of residuals can be informative of how we might improve our model and for assessing whether any modelling assumptions have been met. Using the observations, fitted values and residuals we could construct a series of graphics analyses of the model.

```python
fitted = model.predict(X_train)
resid = y_train - fitted
# Standardise to remove effect of measurement scale
resid = (resid - np.mean(resid))/np.std(resid,ddof = 1)
plt.figure()
```

[6] In classical statistics, one of our assumptions it that the residuals are *normally* distributed.

[7] Small RSS implies the fitted model is closer to the observations.

```
for i in range(3):
  xvar = X_train.iloc[:,i]
  ax = plt.subplot(3, 1, i + 1)
  ax.scatter(xvar, resid)
  ax.set_xlabel(boston.feature_names[i])
  ax.set_ylabel("Residuals")
  ax.hlines([-2, 0, 2], np.min(xvar), np.max(xvar))
plt.show()
```

```
plt.figure()
ax = plt.subplot(3, 1, 1)
ax.scatter(fitted,resid)
ax.set_xlabel('Fitted values')
ax.set_ylabel('Residuals')

ax = plt.subplot(3,1,2)
ax.scatter(fitted,y_train)
ax.set_xlabel('Fitted values')
ax.set_ylabel('Predicted values')

ax = plt.subplot(3, 1,3)
import scipy.stats as stats
stats.probplot(resid,dist = 'norm',plot = ax)
plt.show()
```

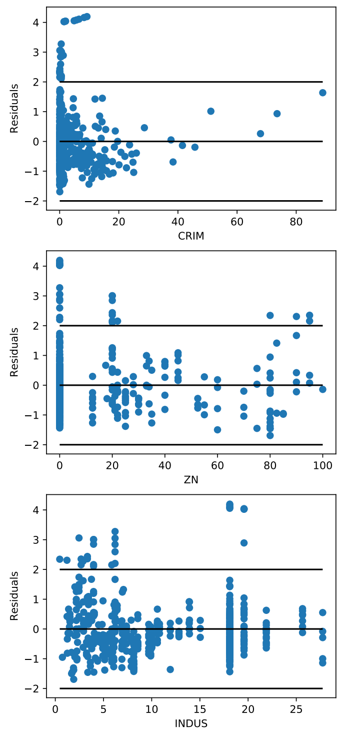


Figure 2.4: Residual analysis against the predictors for the boston housing linear regression model.

2.4 *Scaling data*

Many algorithms benefit from having standardised data.[8]

Why scaling matters

Scaling is crucial for many statistical and machine learning algorithms

- k-means and hierarchical clustering
  - Data units & variance play crucial role in cluster selection
- Using gradient descent optimization
  - Scaled data allows the weights to update at an equal speed
- Scaled data allows the regression coefficients to be compared

There are many common types of rescaling.

Min-max scaling

Min-max scaling is one of the simplest methods for rescaling data. It transforms the data to fit on the range $[0, 1]$. Assume that the original data is x, then the transformed data is defined

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- Numerator: *shifts* the data (resulting in lower bound 0)

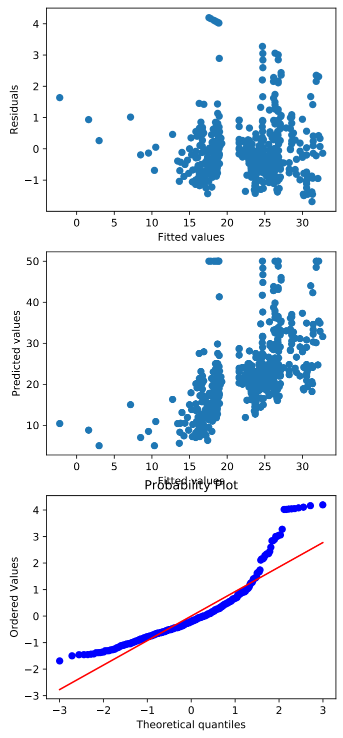


Figure 2.5: Examining fitted values and

- Denominator: *squashes* the data, resulting in an upper bound of 1

Min-max scaling: simulation study

We can examine some of the properties of the rescaling with a simulation study. Suppose we have realisations of random variables drawn from 3 distributions:

- Normal: mean 2, variance 5
- t distribution, with 2 degrees of freedom
- Log normal(1, 1)

We could simulate this data from **numpy**

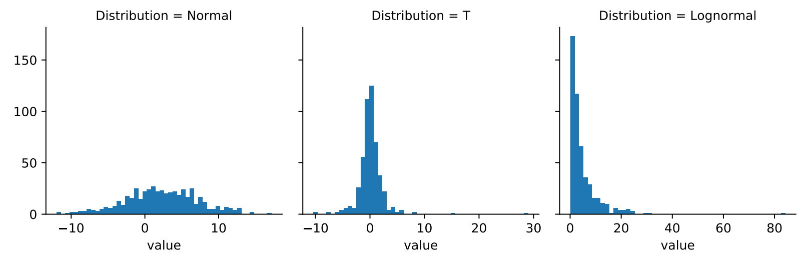
```
import numpy as np
import pandas as pd

np.random.seed(1)

x_n = np.random.normal(2, 5, 500)
x_t = np.random.standard_t(2, 500)
x_ln = np.random.lognormal(1, 1, 500)
df = pd.DataFrame({
  'Normal': x_n,
  'T': x_t,
  'Lognormal': x_ln
})
```

And draw the histogram of each variable

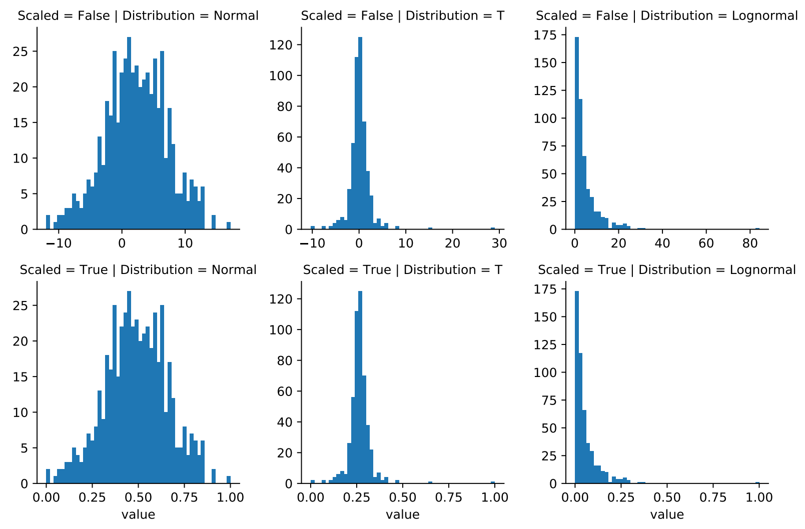
```
import seaborn as sns
import matplotlib.pyplot as plt
df_long = df.melt(var_name='Distribution')
g = sns.FacetGrid(df_long, col='Distribution',sharex=False)
g.map(plt.hist, 'value', bins = 50)
#> <seaborn.axisgrid.FacetGrid object at 0x7fb5992344a8>
plt.show()
```



Applying the transformation to each variable and redrawing the histogram

```
def min_max(x):
  min = np.min(x)
  return (x - min)/(np.max(x) - min)

scaled = df.apply(min_max).melt(
  var_name='Distribution'
)
scaled['Scaled'] = True
df_long['Scaled'] = False
full_data = pd.concat([
  df_long, scaled
  ], axis=0)
g = sns.FacetGrid(
  full_data, col='Distribution',
  row='Scaled', sharex=False,
  sharey=False
)
g.map(plt.hist, 'value', bins = 50)
#> <seaborn.axisgrid.FacetGrid object at 0x7fb56da79080>
plt.show()
```



we can see that min-max scaling doesn't change the profile of the data, only the bounds. The distributional shape remains. However it does change the means and standard deviations.

```
df.apply([np.mean,np.std])
#>          Normal         T  Lognormal
#> mean   2.267184  0.105937   4.837398
#> std    4.946991  2.409213   6.243908
df.apply(min_max).apply([np.mean,np.std])
#>          Normal         T  Lognormal
#> mean   0.488762  0.265530   0.056180
#> std    0.169885  0.061178   0.074298
```

This sort of rescaling is not often used in linear regression problems but is useful in techniques which rely on gradient descent based optimisation routines.

*Min max scaling in **sklearn***

sklearn comes with many preprocessing transformations in the **sklearn.preprocessing** module. To apply the min-max standardisation we can use `MinMaxScaler()`.

```
from sklearn import preprocessing
scaler = preprocessing.MinMaxScaler()
scaler.fit(X_train)
```

We can then create a scaled training set

```
X_train_scaled = scaler.transform(X_train)
X_train_scaled[:1]
#> array([[0.        , 0.18      , 0.06781525]])
```

Z-score standardisation

A common form of standardisation is to rescale each independent variable such that it has zero mean and unit variance.

Given original data x we can define the transfored data

$$x' = \frac{x - \bar{x}}{s}$$

where $\bar{x}$ is the sample mean and s is the sample standard deviation

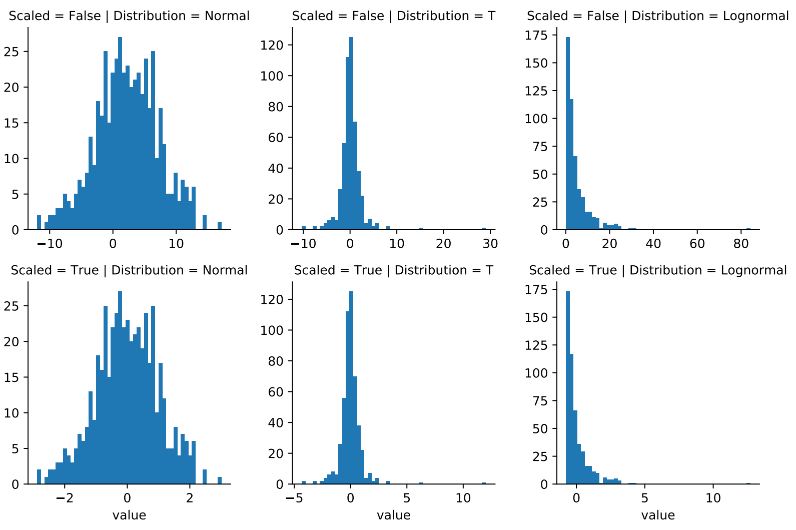
- Numerator: *shifts* the data (resulting in mean 0)
- Denominator: *squashes* the data, resulting in a variance of 1

Z-score standardisation: simulation study

Replicating the same simulation study as before

```
def z_score(x):
  mean = np.mean(x)
  std = np.std(x, ddof=1)
  return (x - mean)/std

scaled = df.apply(z_score).melt(var_name='Distribution')
scaled['Scaled'] = True
full_data = pd.concat([df_long, scaled], axis=0)
g = sns.FacetGrid(
  full_data, col='Distribution',
  row='Scaled', sharex=False,
  sharey=False
)
g.map(plt.hist, 'value', bins=50)
#> <seaborn.axisgrid.FacetGrid object at 0x7fb5614d9d30>
plt.show()
```



we can again see that the profile of the data is unchanged. This sort of scaling is popular in linear models. This won't affect the prediction but makes the size of the coefficients directly comparable.

Z-score standardisation in ***sklearn***

The `sklearn.preprocessing` module has a `StandardScaler()` which can be used to employ this type of zero mean unit variance scaling.

Dividing by two standard deviations

One of the downsides of scaling data by z-scoring is that is not obvious how this should be handled in the case of categorical variables. A paper by Andrew Gelman[9] suggest the use of a rescaling that divides numeric variables by two standard deviations, whilst leaving binary encoded categorical variables untransformed.

[9] `http://www.stat.columbia.edu/~gelman/research/published/standardizing7.pdf`.

The motivation is that this allows the size of coefficients to be directly comparable between numeric and binary variables. The main emphasis of this sort of rescaling is in interpretation of regression models, although it doesn't appear to be particularly widely used.

Two standard deviation scaler in ***sklearn***

There is no dedicated function for this sort of scaling in **sklearn**. However we can define one which can be used in the same way as the others. The way in which we do this is to define a class which inherits from the estimator and transformer classes which define the structure of all of the various estimators and transformers in the package. Doing this we only have to define a fit and a transform method which define the rescaler.

```python
from sklearn.base import BaseEstimator, TransformerMixin

class two_sd_scaler(BaseEstimator, TransformerMixin):

  def fit(self, X, y=None):
    self.stds = 2*np.std(X, axis=0, ddof=1)
    return self

  def transform(self, X, y=None):
    return X/self.stds
```

Fitting a model with scaled data

Having preprocessed the data this way we can not fit a model to it in the same way as before.

```
model2 = linear_model.LinearRegression()
model2.fit(X_train_scaled, y_train)
```

When making predictions on new values we also need to make sure to pass the new values through the same preprocessing step.

```
new_value = np.array(X_train.mean(), ndmin = 2)
new_scaled = scaler.transform(new_value)
pred = model2.predict(new_scaled)
pred
#> array([22.53280632])
```

2.5 *Creating a pipeline*

For any training data set and any data for prediction we will want to apply the same scaling transformation and use the same model. We could create a `sklearn.pipeline.Pipeline()` to organise the steps to creating the estimator.[10]

```
from sklearn.pipeline import Pipeline
model = Pipeline(
  steps = [
    ('preprocess', preprocessing.StandardScaler()),
    ('regression', linear_model.LinearRegression())
  ]
)
```

[10] There is a shorthand `sklearn.pipeline.make_pipeline` which does not allow naming of steps, but is useful in iterating through multiple estimators.

Having created the `Pipeline` object we can now fit as before. Calling `.fit()` now however, will first fit the `'preprocess'` step and then the `'regression'` step. When we predict, the new values will also pass through both stages of our pipeline.

```
model.fit(X_train,y_train)
new_values = np.array(X_train.mean(), ndmin = 2)
model.predict(new_values)
```

2.6 *Preprocessing categorical variables*

With numeric variables, the `StandardScaler()` is often appropriate as it gives everything the same mean and variance. However this transformation is not applicable for categorical variables. Instead we typically want to convert our categorical column into a set of binary dummy variables which inform inclusion in a given category.[11]

[11] We may also want an ordinal transformation when the categorical variables take on a natural order, say small, medium, large.

One hot encoding

One hot encoding will take a categorical feature with K categories and create a 'one of K' encoding scheme. I.e a set of binary variables for each category. Consider the toy data

```
toy = pd.DataFrame({
  'category':['a', 'a', 'b', 'c', 'b']
})
```

We can implement a one hot encoding scheme using the `sklearn.preprocessing.OneHotEncoder`.

```
enc = preprocessing.OneHotEncoder()
enc.fit(toy)
```

With the fitted transformer we can apply the transformation to data

```
enc.transform(toy).toarray()
#> array([[1., 0., 0.],
#>        [1., 0., 0.],
#>        [0., 1., 0.],
#>        [0., 0., 1.],
#>        [0., 1., 0.]])
```

Combining preprocessing steps

We could combine both of the preprocessing steps into a single operation for our `Pipeline` using a `sklearn.compose.ColumnTransformer`

```
toy = pd.DataFrame({
  'numeric': [1., 2., 3., 4., 5.],
  'category': ['a', 'a', 'b', 'c', 'b']
})
```

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

numeric_features = ['numeric']
categorical_features = ['category']

preprocessor = ColumnTransformer(
  transformers=[
    ('num', StandardScaler(), numeric_features),
    ('cat', OneHotEncoder(),  categorical_features)
  ]
)

preprocessor.fit(toy)
```

Applying the transformer will now give the appropriate pre-processing for the different types of variables.

```
preprocessor.transform(toy)
#> array([[-1.41421356,  1.        ,  0.        ,  0.        ],
#>        [-0.70710678,  1.        ,  0.        ,  0.        ],
#>        [ 0.        ,  0.        ,  1.        ,  0.        ],
#>        [ 0.70710678,  0.        ,  0.        ,  1.        ],
#>        [ 1.41421356,  0.        ,  1.        ,  0.        ]])
```

This preprocessing step could then be a step in the pipeline for a regression[12].

```
model = Pipeline(
  steps = [
    ('preprocess', preprocessor),
    ('regression', linear_model.LinearRegression())
  ]
)
```

[12] Pipeline steps can also be pipelines as both single estimators and pipelines are subclasses of general estimator classes. Using this we can build quite complex modelling workflows.

3 Model Assessment and Feature Selection

So far our assessment of a fitted model has been based on comparing it to the data that was used to train the model. Whilst this does have some utility it typically does a poor job at informing us as to how well our model generalises to data it hasn't seen before.

3.1 Resampling techniques

Resampling methods involve repeatedly drawing a sample from a set of training data, fitting a model to each sample and using it to obtain additional information. For example we could learn about the variability of estimated model coefficients or errors. Does this procedure yield consistent inferences?

3.2 Cross validation

Cross validation can be an invaluable tool in assessing the predictive ability of a model, selecting an appropriate level of flexibility or tuning hyperparameters. In cross validation we typically divide our training data into k approximately equally sized groups. Each group in turn is left out as a test set, with the other $k-1$ groups being used as a training set. In model training we could use this to calculate performance metrics for our model, perhaps mean squared error (MSE) in a regression context, or accuracy in classification for each of the held out groups in turn. Since the held out groups are not used for training, we are gaining insight into how the model generalises. The **sklearn.metrics** module provides a number of scoring functions for different model tasks and `sklearn.model_selection.cross_validate` can be used to run a cross validation procedure. We begin by loading some standard libraries

```
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
import pandas as pd
```

and importing the necessary data set

```
boston = load_boston()
X_train, y_train = pd.DataFrame(boston.data, columns = boston.feature_names), boston.target
```

New we set up a modelling Pipeline for the Boston data

```
model = Pipeline(
  steps = [
    ('pre', StandardScaler()),
    ('reg', LinearRegression())
  ]
)
```

To use the metric scoring functions together with cross_validate() we need to wrap them using the make_scorer() function.

```
from sklearn.metrics import mean_squared_error, make_scorer
from sklearn.model_selection import cross_validate
score_fn = make_scorer(mean_squared_error)
scores = cross_validate(model, X_train, y_train,
                              scoring = score_fn,
                              cv = 10)
```

The generated output is a dictionary of timing information for the repeated model fitting processes and the evaluated metric scores on each of the held out cross validation folds. We choose 10 cross validation folds here, meaning that we have 10 hold out estimates of the performance metric.[1]

[1] Research has shown that 10 folds is almost always a good choice.

We could use the cross validation estimates of the performance metrics as a means for comparing different model specifications competency at the same problem. For example it might not be the case that using all 13 predictors for the boston housing data is the best linear regression model that we can find. In the code below we define a train function which wraps up the call to cross_validate(). Doing this means we can very quickly rerun the whole preprocess, model fit and cross validation pipeline for linear models with different predictors as inputs.

```
from itertools import combinations
def train(X):
  return cross_validate(model, X, y_train, \
                scoring = score_fn, \
                cv = 10, \
                return_estimator = True)['test_score']
scores = [train(X_train.loc[:,vars]) for vars in combinations(X_train.columns,12)]
means = [score.mean() for score in scores]
```

In this instance 7 of the 13 possible 12 predictor models give better performance as measure by mean squared error than our full 13 predictor model.

Multiple performance metrics

If we want to measure the performance of our model using multiple metrics, the cross_validate() function allows us to pass a dictionary of scoring functions to be applied to the estimator object for each of the hold out sets. This includes the ability to use our own performance metrics. The make_scorer() utility function expects a function as an argument with signature (y_true, y_pred)[2] Below we define a performance metric which gives us the largest absolute error in prediction, mae_fn(), wrap

[2] Note the names don't matter, but the first argument is the "ground truth" and the second the model prediction.

it as a scorer using `make_scorer()` and pass it to the `cross_validate()` function.

```
import numpy as np
mse = make_scorer(mean_squared_error)
def mae_fn(y_true,y_pred):
  return np.max(np.abs(y_pred - y_true))

mae = make_scorer(mae_fn)
scores = cross_validate(model, X_train, y_train, scoring = {
  'mse' : mse,
  'mae' : mae
}, cv = 10, return_estimator = True)
scores['test_mse'].mean()
#> 34.705255944524836
scores['test_mae'].mean()
#> 14.244172847564489
```

3.3 *Penalised regression*

Generalised linear model parameters, of which linear regression is an example, are typically calculated via maximisation of the model likelihood function. Whilst maximum likelihood estimation has many wonderful statistical properties it can occasionally be unsatisfactory in a regression setting.

- Large variability - if there are a large number of predictors[3], or predictors are highly correlated (not independent) we can get large variance of estimated model parameters. This in turn gives large variance in predictions and suboptimal predictive performance.
- Difficult interpretation - if model interpretation is important we often want a small number of "important" predictors to gain insight into the most relevant relationships between X and y.
- Large model parameters (effect sizes) often also have large variances.

[3] X variables.

One solution to this is to choose a subset of explanatory variables. It is often infeasible to check all possible model configurations (all possible variable subsets) using cross validation.[4]

[4] In the Boston housing example, using only the 13 available features, without considering transformations or additional engineered features we would need to fit 8190×10 linear regression models to get all the cross validation estimates.

Even when it is feasible, this solution would not address the issue of high variance, or correlated predictors. We would still remain with large effect sizes that require shrinkage.

The idea behind penalised regression techniques is to address the issue by introducing a penalty to large estimates of coefficients. The success of such techniques is founded in the bias-variance trade-off. By introducing a penalty we also introduce a bias into the estimator for coefficients.[5] However by penalising the large coefficients we are aiming to reduce variance. Since total error is a function of both bias and variance we hope that the induced error from the bias is smaller than the reduction achieved in variance. When this is the case the resultant predictive power of our model will have better predictive power due to smaller total error.

[5] Bias here meaning that the expectation of the estimated value is different from the unknown true value.

Instead of maximising a likelihood function directly (equivalently minimising the negative log likelihood) we typically minimise a function of the form

$$M(\theta) = \overset{\text{Loss function}}{L(\theta\,|\,X)} + \overset{\text{Penalty term}}{\lambda P(\theta)}$$

- The loss function $L(\theta\,|\,X)$ is typically proportional to the negative log likelihood (e.g RSS in linear regression) and controls movement of the model towards the data points
- $P(\theta)$ penalises "less realistic" parameter values
- λ controls the trade-off between the two

λ - the regularization parameter

We have introduced an additional hyper-parameter into our model estimation process. Clearly choice of λ is important since it controls the trade-off between model fit and penalty.

- If $\lambda = 0$, the loss function dictates the model fit and we get standard linear regression (zero bias but potentially large variance).
- If $\lambda = \infty$, the penalty effectively forces all coefficients to be zero and we get a null model (zero variance, but very high bias).
- λ effectively controls the bias variance trade-off. Ideally, a good choice of λ will introduce a small bias for a large reduction in variance.

LASSO (`Least Absolute Shrinkage and Selection Operator`)

Lasso regression is an example of such a technique. It follows the above prescription with a penalty function defined as

$$P(\beta) = \sum_{i=1}^{n} |\,\beta_i\,|.$$

That is, the sum of the absolute values of the coefficients. This has the consequence of penalising regression coefficients that are further away from 0, favouring smaller values of β_i, (shrinkage), reducing the variance. When β_i is sufficiently penalised it is set to 0, effectively removing it from the model fit (selection).

We can use cross validation as a means for selecting the value of our hyper-parameter λ. **sklearn** implements a `LassoCV` estimator which will automatically take care of trialling different values of λ and finding the best one using cross validation in an efficient manner. As with all other model estimators, we could build a pipeline for training our model. It is important to note that rescaling is now very important. Since β is a function of measurement scale and dictates the contribution to the penalty, we want to standardise predictors such that they are on a common measurement scale. For example by using the `StandardScaler()` to give all predictors 0 mean unit variance.

```
from sklearn.linear_model import LassoCV
model = Pipeline([
  ('pre', StandardScaler()),
  ('mod', LassoCV(cv = 10))
])
model.fit(X_train,y_train)
```

From our fitted model we can extract the coefficients estimated from the final step in our pipeline

```
model.steps[-1][1].coef_
#> array([-0.49642878,  0.53758696, -0.05979688,  0.64601683, -1.35784426,
#>         2.89535011, -0.        , -2.10431451,  0.52531783, -0.28422681,
#>        -1.86040633,  0.72184225, -3.72077045])
```

We can see that some coefficients have been set to be 0.

Method	λ_1	λ_2
Linear	0	0
Ridge	0	$\neq 0$
Lasso	$\neq 0$	0
Elastic	$\neq 0$	$\neq 0$

Table 3.1: A summary of what type of model is fit given penalty parameters using a penalized regression model.

3.4 Other penalised regression techniques

There are other popular techniques which operate in a similar way but vary by choice of the penalty function. In particular

- Ridge regression - see **sklearn.linear_model.RidgeCV**, which has penalty based on squared coefficients, $P(\beta) = \sum_i \beta_i^2$. This penalty function acts as a more aggressive shrinkage operation (hence often greater variance reduction) but has the consequence that $\beta_i \neq 0 \forall i$, i.e no subset selection.
- Elastic net - see **sklearn.linear_model.ElasticNetCV**, which mixes both lasso and ridge penalties, $P(\beta) = \lambda_1 \sum_i |\beta_i| + \lambda_2 \sum_i \beta_i^2$ which aims to get the best of both.

4 Classification

In many situations the response is qualitative rather than quantitative. In this chapter we will consider some techniques for predicting categorical[1] variables, a process known as classification. Predicting a categorical response typically involves assigning the observation to a category, classifying that observation.

[1] Often qualitative variables are referred to as categorical.

4.1 Logistic regression

Consider the breast cancer data from the **sklearn** package. The data relates 30 predictor variables to the diagnosis of whether or not a tumor was malignant or benign, recorded as 0 and 1 respectively. To begin we will load everything we need to build a model `Pipeline` for this task.

```
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
cancer = load_breast_cancer()
X_train, y_train = cancer.data, cancer.target
cancer.target_names
#> array(['malignant', 'benign'], dtype='<U9')
```

With logistic regression we are fitting a model of the form

$$\log\left(\frac{\pi(X)}{1-\pi(X)} = \beta_0 + \beta X\right)$$

where $\pi(X)$ is the probability $Pr(Y = 1 \mid X)$. The left hand side of this equation is called the *log-odds*.

As with the other modelling techniques that we have come across so far we can add the steps to a `Pipeline` and preprocess and fit in one go.

```
model = Pipeline([
  ('pre', StandardScaler()),
  ('logis', LogisticRegression(class_weight = 'balanced'))
])
model.fit(X_train, y_train)
```

4.2 *Quantifying error*

The performance metrics that we used for regression problems, such as mean square error, are no longer appropriate since we don't observe the true probability of belonging to a certain category. Instead we might examine, among other things,

- Accuracy - The proportion of correct classifications made by the model.
- Precision - The proportion of correct positive classifications of those predicted as positive. (Typically this makes most sense when one of the classes is an event of interest.)
- Recall - The proportion of correct positive classifications of those that were truly positive in the reference data.[2]

[2] Recall is often called sensitivity, true positive rate or probability of event detection.

The **sklearn.metrics** module has a number of functions for quantifying the efficacy of a classification model

```
from sklearn.metrics import accuracy_score, precision_score, recall_score
y_pred = model.predict(X_train)
accuracy_score(y_train,y_pred)
#> 0.984182776801406
precision_score(y_train,y_pred,pos_label=0) # tp/(tp + fp)
#> 0.9810426540284360
recall_score(y_train,y_pred,pos_label=0) # tp/(tp + fn)
#> 0.9764150943396226
```

As with our regression examples, typically we are not interested in how the performance metrics measure the ability of the model to classify the training data. Instead we want to know how well it generalises. We could use cross validation to repeatedly separate the available data into training and validation sets to get a better picture.

```
from sklearn.model_selection import cross_validate
from sklearn.metrics import make_scorer
import pandas as pd

acc = make_scorer(accuracy_score)

def precision(y_true,y_pred):
  return precision_score(y_true,y_pred,pos_label = 0)

def recall(y_true,y_pred):
  return recall_score(y_true, y_pred, pos_label = 0)

prec = make_scorer(precision)
rec = make_scorer(recall)
output = cross_validate(model,X_train,y_train,scoring={
  'acc' : acc,
  'prec' : prec,
  'rec' : rec
}, cv = 10, return_train_score=False)
```

```
pd.DataFrame(output).mean()
#> fit_time       0.028558
#> score_time     0.003257
#> test_acc       0.970113
#> test_prec      0.954378
#> test_rec       0.966883
#> dtype: float64
```

From this we can ascertain that the model is doing relatively well at classifying the tumors.

4.3 *Linear Discriminant Analysis (LDA)*

Logistic regression directly models $Pr(Y = k \mid X = x)$ using the logistic function but is typically used when there are only two categories in response. LDA classifies an observation into one of K classes $K \geq 2$

Remember X is the random variable, x is a observed value of that random variable.

$$Pr(Y = k \mid X = x) = \frac{\pi_k f_k(x)}{\sum_{l=1}^{K} \pi_l f_l(x)} \tag{4.1}$$

where

- π_k is the prior probability that an observation comes from class k;
- $f_k(X) = Pr(X = x \mid Y = k)$, the density function of X for an observation from class k.

The goal is to find estimates of π_k and $f_k(X)$ from the training data to plug into equation (4.1) to classify a future observation.

$f_k(X)$ is assumed to have a Gaussian distribution[3]. That is we are assuming that the distribution of each of the predictors for observations in a given class are approximately normal with mean μ_k and variance σ_k^2. In particular LDA makes the assumption that there is common variance for a predictor across all classes

[3] The Gaussian distribution is also known as the Normal distribution.

$$\sigma_1 = \ldots = \sigma_k. \tag{4.2}$$

We classify a response for an observation X to the group k which maximises ((4.1)).

For multiple predictors we instead make the assumption that within each response group the predictors come from a multivariate Gaussian with mean vector μ_k and common covariance matrix Σ.

LDA takes a set of p predictors and reduces them to dimension $k - 1$ (the number of response categories) via a set of k discriminant function which are linear in x[4].

[4] Hence the name.

We can fit a model using LDA fit in Python using `sklearn.discriminant.LinearDiscriminantAnalysis`. Performing LDA on the `diabetes` data set.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

model = Pipeline([
  ('pre', StandardScaler()),
  ('lda', LinearDiscriminantAnalysis())
])
model.fit(X_train, y_train)
#> Pipeline(memory=None,
#>          steps=[('pre',
#>                  StandardScaler(copy=True, with_mean=True, with_std=True)),
#>                 ('lda',
#>                  LinearDiscriminantAnalysis(n_components=None, priors=None,
#>                                             shrinkage=None, solver='svd',
#>                                             store_covariance=False,
#>                                             tol=0.0001))],
#>          verbose=False)
model.named_steps['lda'].priors_
#> array([0.37258348, 0.62741652])
model.named_steps['lda'].means_
#> array([[ 0.94734027,  0.53877588,  0.96370009,  0.92003111,  0.46529456,
#>          0.77410727,  0.90364908,  1.00779292,  0.42887995, -0.01665905,
#>          0.73595579, -0.01077503,  0.72169029,  0.71143247, -0.08696505,
#>          0.380218  ,  0.32925896,  0.52950663, -0.00846312,  0.10118291,
#>          1.00758521,  0.5929117 ,  1.01596866,  0.95226693,  0.54692472,
#>          0.76692406,  0.85596015,  1.02979135,  0.54021502,  0.42028107],
#>        [-0.56256621, -0.31994534, -0.57228129, -0.54634901, -0.27630937,
#>         -0.45969395, -0.53662074, -0.59846526, -0.25468501,  0.00989277,
#>         -0.43703817,  0.00639862, -0.42856678, -0.4224753 ,  0.05164311,
#>         -0.22578772, -0.19552633, -0.31444091,  0.00502572, -0.06008621,
#>         -0.59834192, -0.35209322, -0.60332033, -0.56549185, -0.32478442,
#>         -0.45542829, -0.50830127, -0.61152876, -0.32079995, -0.24957868]])
model.named_steps['lda'].coef_
#> array([[ 1.44833568e+01, -3.68957962e-01, -1.08865782e+01,
#>         -2.11087855e+00, -2.24784150e-02,  4.20809820e+00,
#>         -2.10328313e+00, -1.56846323e+00, -5.31387551e-02,
#>         -4.43195602e-03, -2.27635626e+00,  7.03616573e-02,
#>          8.59307158e-01,  7.92601536e-01, -8.98376566e-01,
#>         -2.19353267e-02,  2.03117873e+00, -1.23061216e+00,
#>         -2.64794590e-01,  3.56875236e-01, -1.78035220e+01,
#>         -8.30445328e-01,  1.54420617e+00,  1.08656797e+01,
#>         -2.33917449e-01, -1.99413370e-01, -1.50083483e+00,
#>         -5.75986366e-01, -6.50095283e-01, -1.46687628e+00]])
```

- Prior probabilities - if no other information is given they are estimated to be the proportion of each response in the training data.[5]
- Group means - the means of each covariate within the response groups.
- Coefficients of linear discriminants - the linear combination of covariates used to form the LDA decision rule.

[5] Prior represent the initial belief we have before fitting the model.

```
output = cross_validate(model,X_train,y_train,scoring={
  'acc' : acc,
  'prec' : prec,
  'rec' : rec
}, cv = 10, return_train_score=False)
```

```
pd.DataFrame(output).mean()
#> fit_time       0.008946
#> score_time     0.003537
#> test_acc       0.956046
#> test_prec      0.990238
#> test_rec       0.891775
#> dtype: float64
```

Quadratic Discriminant Analysis (QDA)

QDA is a more flexible version of LDA where the assumption of shared variance is relaxed. That is the Gaussian distribution of predictors within each group has its own covariance matrix Σ_k.[6]

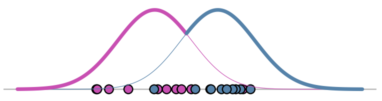
The `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis` class is used in the same way

[6] QDA gets its name from the fact that the discriminant function is now quadratic in x. This allows decision boundaries to become quadratic in predictor space as opposed to the linear boundaries imposed in LDA. See figure 4.1.

```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

model = Pipeline([
  ('pre', StandardScaler()),
  ('qda', QuadraticDiscriminantAnalysis())
])
model.fit(X_train, y_train)
#> Pipeline(memory=None,
#>          steps=[('pre',
#>                  StandardScaler(copy=True, with_mean=True, with_std=True)),
#>                 ('qda',
#>                  QuadraticDiscriminantAnalysis(priors=None, reg_param=0.0,
#>                                                store_covariance=False,
#>                                                tol=0.0001))],
#>          verbose=False)
```

See figure 4.1 for a comparison of how the model affects the decision boundary for a number of classification models on a simulated data set.



Difference between LDA and QDA

- LDA is a much less flexible classifier due to common variance assumption.
- For k categories with p predictors LDA has to estimate kp linear coefficients. In QDA estimating the covariance matrices require $kp(p-1)/2$. This quickly adds up to a lot of parameters.
- LDA classifier has substantially lower variance so can give better prediction performance.
- If the common variance assumption is poor it can result in high bias.
- LDA tends to be better for smaller training data.
- QDA better when constant variance assumption clearly untenable.

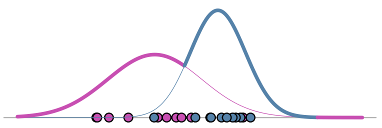


Figure 4.1: Differences between LDA (top) and QDA (bottom). LDA assumes the standard deviation between groups is the same.

4.4 *K-nearest neighbours (KNN)*

Theoretically a KNN classifier is a simple concept. Given a test observation, x_0, we identify the K points in the training data that are closes to x_0, denote this set of points as N_0. We then estimate the probability that the response belongs to group j

$$Pr(Y = j \,|\, X = x_0) = \frac{1}{K} \sum_{i \in N_0} I(y_i = j)$$

That is we estimate the probability of being in group j based on the proportion of the K nearest neighbours that belong to that group and classify according to whichever class has the highest probability. All that remains is to choose the number of neighbours K.

- A small K gives a very flexible decision boundary leading to low bias but high variance and the potential to overfit.
- A large K makes the decision boundary more linear, increasing the bias for a reduction in variance with the potential to underfit.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_wine

wine = load_wine()
X_train, y_train = wine.data, wine.target

model = Pipeline([
  ('pre', StandardScaler()),
  ('knn', KNeighborsClassifier())
])

model.fit(X_train,y_train)
```

Tuning the hyperparameter K

We want to find the value of K which yields the best performance. We can quickly scan a range of parameter values in our model `Pipeline` using grid search cross validation, `GridSearchCV`.

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix
clf = GridSearchCV(model, param_grid= {
  'knn__n_neighbors' : [1,3,5,10,15,20,25,30,35,40]
}, cv = 10, iid = False, return_train_score=False)
clf.fit(X_train,y_train)
```

The best chosen parameters for the scoring function are stored in the `.best_params_` attribute. Note that we could specify the specific scoring function to use, but here we stick with the default accuracy measure.

```
clf.best_params_
#> {'knn__n_neighbors': 35}
```

```
clf.best_score_
#> 0.9777777777777779
pd.DataFrame(clf.cv_results_)[['params','mean_test_score','rank_test_score']]
#>                         params  mean_test_score  rank_test_score
#> 0   {'knn__n_neighbors': 1}         0.943464               10
#> 1   {'knn__n_neighbors': 3}         0.949020                9
#> 2   {'knn__n_neighbors': 5}         0.966013                7
#> 3  {'knn__n_neighbors': 10}         0.971895                2
#> 4  {'knn__n_neighbors': 15}         0.966340                5
#> 5  {'knn__n_neighbors': 20}         0.966667                4
#> 6  {'knn__n_neighbors': 25}         0.966340                5
#> 7  {'knn__n_neighbors': 30}         0.966013                7
#> 8  {'knn__n_neighbors': 35}         0.977778                1
#> 9  {'knn__n_neighbors': 40}         0.971895                2
```

A confusion matrix is a handy way to quickly see how well the model is doing at classifying each individual class.

```
confusion_matrix(y_train,clf.predict(X_train))
#> array([[59,  0,  0],
#>        [ 2, 67,  2],
#>        [ 0,  0, 48]])
```

Note that whilst we have introduced K nearest neighbours in the context of classification problems, it also has utility in regression problems where prediction is typically taken to be the average of the K surrounding points. See `sklearn.neighbors.KNearestRegressor`.

4.5 *Decision trees*

Decision trees are a non-parametric approach to supervised learning used for both classification and regression. It aims to learn simple decision rules inferred from the data. One of the advantages of this rule based approach is that it yields a model which is simple to understand and interpret, including for non statisticians. In addition they typically require little data preparation, i.e scaling predictors is not necessary.

Intuition

Decision trees are found by recursively partitioning data. The idea is I search amongst all the variables to find the point which best splits the data into two parts. In each of these two parts I then do the same again, continuing until I hit some stopping criteria.

In a classification problem the way in which we decide on "best split" is typically based on the homogeneity of data in the nodes. For example imagine there are two variables, age and income, which are being used to try to predict whether a customer makes a purchase. If the available training data showed that 95% of people over 30 made the purchage, the split would be made here and age becomes the top node in my decision tree. You could describe the subset of people over 30 as 95% pure (95% of people have the same outcome). There are many common ways to measure impurity across the tree, such as Gini for quantifying homogeneity in the nodes. We are essentially trying to minimise impurity with each partition.

In theory we could recursively partition the data until each observation is alone in a node, but this would be a terrible case of over fitting so we often prune the tree by employing some penalty on complexity or having a terminating condition based on the maximum allowed depth of the tree.

```
import sklearn.tree as tree
from sklearn.metrics import confusion_matrix
model = tree.DecisionTreeClassifier()
model.fit(X_train,y_train)
```

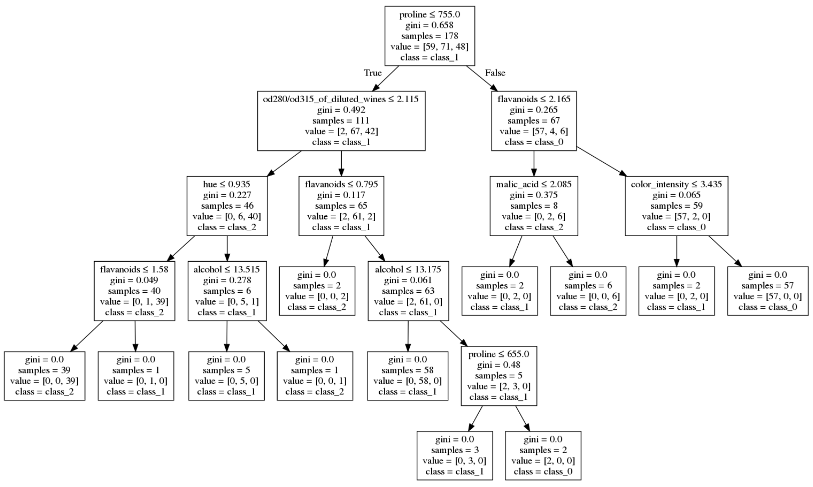
After fitting a model we can examine diagnostics such as a confusion matrix and cross validation estimates of performance metrics for comparison to other models that we fit.

```
confusion_matrix(y_train, model.predict(X_train))
#> array([[59,  0,  0],
#>        [ 0, 71,  0],
#>        [ 0,  0, 48]])
```

```
from sklearn.model_selection import cross_validate
val = cross_validate(model,X_train,y_train, cv = 10)
val['test_score'].mean()
#> 0.8647058823529411
```

The **graphviz** package can be useful for visualising single tree models.

```
import graphviz
dot_data = tree.export_graphviz(model,out_file = None,
  feature_names=wine.feature_names,
  class_names=wine.target_names,special_characters=True)
graph = graphviz.Source(dot_data)
graph.render(filename='tree',format='png')
```



proline ≤ 755.0 gini = 0.658 samples = 178 value = [59, 71, 48] class = class_1
True
False
od280/od315_of_diluted_wines ≤ 2.115 gini = 0.492 samples = 111 value = [2, 67, 42] class = class_1
flavanoids ≤ 2.165 gini = 0.265 samples = 67 value = [57, 4, 6] class = class_0
hue ≤ 0.935 gini = 0.227 samples = 46 value = [0, 6, 40] class = class_2
flavanoids ≤ 0.795 gini = 0.117 samples = 65 value = [2, 61, 2] class = class_1
malic_acid ≤ 2.085 gini = 0.375 samples = 8 value = [0, 2, 6] class = class_2
color_intensity ≤ 3.435 gini = 0.065 samples = 59 value = [57, 2, 0] class = class_0
flavanoids ≤ 1.58 gini = 0.049 samples = 40 value = [0, 1, 39] class = class_2
alcohol ≤ 13.515 gini = 0.278 samples = 6 value = [0, 5, 1] class = class_1
gini = 0.0 samples = 2 value = [0, 0, 2] class = class_2
alcohol ≤ 13.175 gini = 0.061 samples = 63 value = [2, 61, 0] class = class_1
gini = 0.0 samples = 2 value = [0, 2, 0] class = class_1
gini = 0.0 samples = 6 value = [0, 0, 6] class = class_2
gini = 0.0 samples = 2 value = [0, 2, 0] class = class_1
gini = 0.0 samples = 57 value = [57, 0, 0] class = class_0
gini = 0.0 samples = 39 value = [0, 0, 39] class = class_2
gini = 0.0 samples = 1 value = [0, 1, 0] class = class_1
gini = 0.0 samples = 5 value = [0, 5, 0] class = class_1
gini = 0.0 samples = 1 value = [0, 0, 1] class = class_2
gini = 0.0 samples = 58 value = [0, 58, 0] class = class_1
proline ≤ 655.0 gini = 0.48 samples = 5 value = [2, 3, 0] class = class_1
gini = 0.0 samples = 3 value = [0, 3, 0] class = class_1
gini = 0.0 samples = 2 value = [2, 0, 0] class = class_0

Figure 4.2: Visualisation of a decision tree model.

5

More Advanced

5.1 Random forest

Single decision trees, whilst simple and intuitive, often perform poorly for predictive models. Either due to complex trees with high variance and over fitting, or trees which are too simple which gives high bias but under fit the data.

Random forest models are an ensemble learning method applicable for both regression and classification tasks. They offer an improvement over single trees by growing many trees on bootstrap[1] samples of the data and aggregating prediction over all trees. In addition each split in the tree growing process considers only a random subset of predictor variables. This often provides strong performance as aggregating over complex trees has the effect of reducing the variance, whilst a random subset of predictors at each split de-correlates trees, further reducing variance in the aggregate. Random forests are fairly robust to over fitting, so in practice we typically keep adding trees until our estimated performance metrics settle down.

THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS
Figure 5.1: https://xkcd.com/1838/

[1] Bootstrap samples are samples from the original data of the same dimension, sampled with replacement.

Below we load everything that we need for training and testing a `RandomForestClassifier`[2]

[2] See also `RandomForestRegressor`.

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix
```

For this example we will use a random forest to classify images of hand written digits given the grey scale colour intensities of each pixel as features

```python
digits = load_digits()
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size = 0.2)
clf = RandomForestClassifier(oob_score = True)
```

In addition the cross validation estimates of model performance we break our data further to exclude a dedicated final test set from the model fitting and cross validation procedure. As before we will use a grid search on model hyperparameters to try to find a good model. The parameters here

- `n_estimators` - the number of trees to grow in the forest. Typically each tree is complex and unpruned.

- max_features - the proportion of the feature set to consider at each node split.

```
param_search = {
  'n_estimators': [50, 100, 500], # of trees in forest
  'max_features': [0.8, 0.5, 0.3]
}
output = GridSearchCV(clf, param_search, \
                              cv = 10, \
                              return_train_score = False)
output.fit(X_train, y_train)
```

We can examine the performance of the model through the cross validation estimates of the default scoring function[3]. The best parameters and best accuracy score in the grid search are

[3] The default is accuracy, see previous section for choosing a different scoring function.

```
output.best_params_
#> {'max_features': 0.3, 'n_estimators': 500}
output.best_score_
#> 0.9749514374514374
```

We can then see how well it does on the dedicated test set that it has never seen before.

```
output.score(X_test, y_test)
#> 0.9694444444444444
```

This is a pretty high accuracy rate for classifying hand written digits, especially considering how grainy the images are. Below are the reference images together with ground truth values and prediction for three digits in the test set.

```
import matplotlib.pyplot as plt
val = output.predict(X_test[:3])
plt.figure(figsize = [10, 7.5])
plt.gray()
for i in range(3):
  ax = plt.subplot(1, 3, i+1)
  ax.set_title("Real: {} Predicted: {}".format(val[i], y_test[i]))
  ax.matshow(X_test[i].reshape(8, 8))
  ax.set_xticks(())
  ax.set_yticks(())
plt.show()
```

5.2 *Support vector machines*

A support vector classifier is a classifier which predicts the class of an observation based on which side of a hyper plane it lies on. Essentially a hyper plane is drawn through predictor space to try to separate most of the training observations into the two classes.[4] A support vector classifier seeks a linear decision boundary and hence on its own is poor in many situations. A support vector machine hopes to find non-linear decision boundaries by expanding the feature space.[5] The support vector machine is an extension

[4] Multinomial class predictions can be made by using a one against the rest rule for the hyperplane.

[5] Similar to the idea behind adding an x^2 term in linear regression.

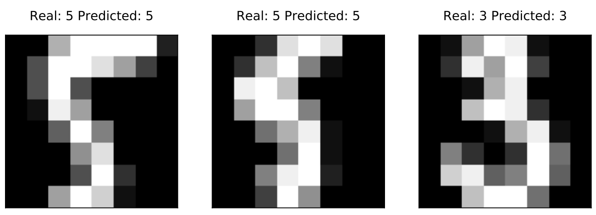
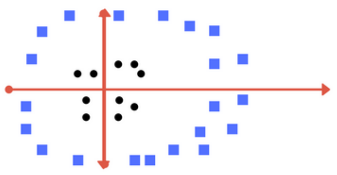


Figure 5.2: Predicting handwritten digits using a random forest classifier.

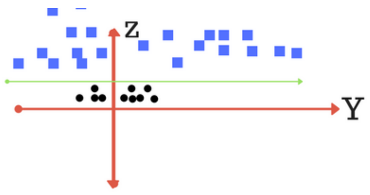
of the support vector classifier that does this space expansion in a particular way, namely using kernels. It is some, essentially arbitrary, function to quantify the similarity between two vectors of predictors. Common kernels are linear, polynomial and radial and are just a way to designate closeness between observations. The kernel tells the model how strongly x should influence the prediction of a point x^* based on how close they are. By expanding the feature space we can fit non-linear hyperplanes to separate the classes.

Intuition

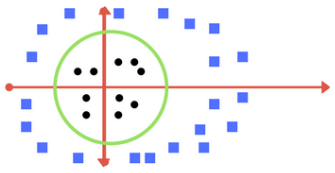
Imagine we want to classify points into black and blue classes given the data as shown below



There is no way with a straight line that we could separate the two classes. However if we added a new feature say $z = x^2 + y^2$ and generate a plot in the z-y axis we see that a separation can be made.



When we transform that line back to the original axes it maps to a circular boundary



Essentially the kernel is used to expand the original space into a higher dimensional space, hopefully taking a non separable problem to a separable one. So the kernel does the job of finding those features (like $x^2 + y^2$ in our example).

SVM's are often considered one of the best, most stable classifiers.

```
from sklearn.svm import SVC
clf = SVC()
params = {
  'gamma': [0.0001, 0.001, 0.01, 0.1], # small gamma should let data have stronger influence
  'C': [0.5, 1, 1.5] # small C allow more misclassification in support vectors
}
output = GridSearchCV(clf, params, cv = 10, return_train_score = False)
output.fit(X_train,y_train)
```

```
import pandas as pd
pd.DataFrame(output.cv_results_)[['param_C','param_gamma','mean_test_score']]
#>    param_C param_gamma  mean_test_score
#> 0      0.5      0.0001         0.963826
#> 1      0.5       0.001         0.986781
#> 2      0.5        0.01         0.338918
#> 3      0.5         0.1         0.105779
#> 4        1      0.0001         0.974267
#> 5        1       0.001         0.990958
#> 6        1        0.01         0.776617
#> 7        1         0.1         0.112738
#> 8      1.5      0.0001         0.977744
#> 9      1.5       0.001         0.990958
#> 10     1.5        0.01         0.795406
#> 11     1.5         0.1         0.115516
output.score(X_test,y_test)
#> 0.9833333333333333
confusion_matrix(output.predict(X_test),y_test)
#> array([[42,  0,  0,  0,  0,  0,  0,  0,  0,  0],
#>        [ 0, 46,  0,  0,  0,  0,  0,  0,  2,  0],
#>        [ 0,  0, 27,  0,  0,  0,  0,  0,  0,  0],
#>        [ 0,  0,  0, 38,  0,  0,  0,  0,  0,  0],
#>        [ 0,  0,  0,  0, 29,  0,  0,  0,  0,  0],
#>        [ 0,  0,  0,  0,  0, 38,  0,  0,  0,  0],
#>        [ 0,  0,  0,  0,  0,  1, 40,  0,  0,  0],
#>        [ 0,  0,  0,  1,  0,  0,  0, 28,  0,  0],
#>        [ 0,  0,  0,  0,  1,  0,  0,  0, 29,  0],
#>        [ 0,  0,  0,  0,  0,  1,  0,  0,  0, 37]])
```

We could also plot the profile of our classifiers hyperparameters with respect to the scoring function. This would give some insight as to which hyperparameters we would try next in order to iteratively improve our model fit.

```
res = pd.DataFrame(output.cv_results_)
color_map = {
  0.5: 'r',
  1.0: 'g',
  1.5: 'b'
}
plt.figure()
for c in res.param_C.unique():
  sub = res[res.param_C == c]
  g = pd.to_numeric(sub['param_gamma'])
  m = sub['mean_test_score']
  s = sub['std_test_score']
  plt.plot(g,m, color = color_map[c], label = c)
  plt.fill_between(g,m-s,m+s,alpha = 0.1, color = color_map[c]

plt.xscale('Log')
plt.legend()
plt.show()
```

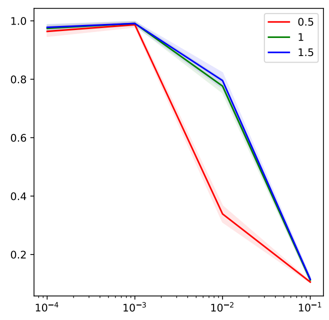


Figure 5.3: Profiling accuracy against the hyperparameters for SVM classifier.

6 Clustering

Cluster analysis is a term that covers a wide range of numerical techniques for uncovering groups (or clusters) of observations in a dataset. For example, with micrarray analysis, we might want to cluster genes that have similar profiles.

These sorts of techniques are relatively easy when there are only a few dimensions. For high dimensional spaces we start to suffer from the curse of dimensionality.[1] This is because typical distance metrics such as Euclidean distance become less meaningful.

[1] https://en.wikipedia.org/wiki/Curse_of_dimensionality.

With cluster analysis we are aiming to find gropus of observations that are "similar", so it is required that we define what "similar" means. Typically this is done by defining a distance function between two observations, where a small distance indicates that two observations are similar. For every pair of observations, x_i and x_j we need to define a corresponding distance, $d_{i,j}$. Since we have n observations, this corresponds to an $n \times n$ distance matrix D.

Theoretically we can choose any distance function that defines a metric:

- All distances are positive, i.e. $d_{ij} \geq 0$
- The distance of an observation with itself is 0, i.e. $d_{ii} = 0$
- A zero distance implies that the observations are equal, i.e. $d_{ij} = 0$ implies that $x_i = x_j$
- The distance between observation i and j is the same as between j and i, i.e. $d_{ij} = d_{ji}$

6.1 Euclidean distance (L_2 norm)

Euclidean distance, or L_2 norm is the sum of squared distances between each variable for a pair of observations.

$$d_{ij} = d(x_i, x_j) = \sum_{k=1}^{p} (x_{i,k} - x_{j,k})^2$$

Or in matrix notation

$$d_{ij} = d(x_i, x_j) = \sqrt{(x_i - x_j)^T (x_i - x_j)}$$

In the absence of prior knowledge over good choice of distance function this is a reasonable default measure. It is almost certainly the most commonly used distance function in applied cluster analysis.

6.2 Manhattan distance (L_1 norm)

The Manhattan distance, or L_1 norm is the sum of absolute distances between the variables

$$d_{ij} = d(x_i, x_j) = \sum_{k=1}^{p} |x_{i,k} - x_{j,k}|$$

The Manhattan distance often turns out to be a good choice when there is sparsity in the features.

6.3 Mahalanobis distance

Where variables are highly correlated, there is often advantage in considering the Mahalonibis distance. This is essentially similar to the Euclidean metric but rescaled by the covariance matrix.

$$d_{ij} = (x_i - x_j)^T S^{-1} (x_i - x_j)$$

where S is the *covariance* matrix.

- If S is the identity matrix, i.e. the diagonals are 1, then mahalanobis reduces to the Euclidean distance
- If S is diagonal, i.e. the correlation between the variables is 0, then

$$d_{ij} = d(x_i, x_j) = \sum_{k=1}^{p} \frac{(x_{i,k} - x_{j,k})^2}{s_i^2}$$

6.4 Heirarchical clustering

Given that we can measure distance between pairs of observations we can begin our clustering. Heirarchical cluster analysis is an approach to finding clusters in a nested, or heirarchical structure. Heirarchical clustering typically falls into one of two strategies:

- Agglomerative (Bottom up)
- Divisive (Top down)

Agglomerative strategies appear to be more common. An agglomerative strategy to cluster analysis could be defined as follows:

1. **Initialise**: Start with n clusters, C_1 , $C_2, \ldots, C_n$, with C_i containing just the single, p dimensional observation x_i
2. Find the minimum distance, d_{ij} between the clusters and combine the clusters
3. The number of clusters has now decreased by one
4. If there is more than a single cluster remaining, return to step 2

One of the key steps here requires definition of minimum distance between clusters. In order to put this algorithm into practice, in addition to requiring distances between observations, we also need to be able to calculate distance between clusters.

Single linkage

Single linkage defines the distance between two clusters as the minumum distance between observations in the clusters

$$d_{c_1,c_2} = \min_{i \in c_1, j \in c_2} d_{ij}$$

The single linkage method tends to perform well in situations where data are not globular. However it also tends to lead to a rich get richer type attitude leading to very unbalanced final cluster sizes. An additional advantage of single linkage is that it is very efficient to compute, leading to a preferance in the case of large numbers of observations.

Complete Linkage

Complete linkage gives the distance between two clusters as the maximum distance between observations

$$d_{c_1,c_2} = \max_{i \in c_1, j \in c_2} d_{ij}$$

This is in effect the opposite of single linkage. It tends to perform well in the presence of lots of noise, however it tends to have bias towards globular structures and tends to break large clusters.

Average linkage

The average distance between observations in clusters

$$d_{c_1,c_2} = \frac{1}{n_{c_1} n_{c_2}} \sum_{c_1} \sum_{c_2} d_{ij}$$

Tends to exhibit similar properties to complete link age, but is more expensive to compute. It is not as popular as other choices in practice.

Worked example

To describe how agglomerative heirachical clustering works in practice we shall work through a very small toy example by hand.

```
import pandas as pd
import numpy as np

example = pd.DataFrame({
  'x': [7, 2, 1, 4],
  'y': [4, 4, 2, 8]
})

example
#>    x  y
#> 0  7  4
#> 1  2  4
#> 2  1  2
#> 3  4  8
```

If we were to use the Manhattan distance between observations, this would yield a distance matrix of

```
def manhattan(x, y):
  return np.sum(np.abs(x - y))

D = pd.DataFrame(np.array([
  [manhattan(
    example.iloc[i,:],
    example.iloc[j,:]
  ) if j <= i else 0 for j in range(4)] for i in range(4)
]),
index=['c'+str(i) for i in range(1,5)],
columns=['c'+str(i) for i in range(1,5)])
D
#>     c1  c2  c3  c4
#> c1   0   0   0   0
#> c2   5   0   0   0
#> c3   8   3   0   0
#> c4   7   6   9   0
```

If we were to use the single linkage, then we look for the minimum, non-zero distance between these clusters. In this case that is 3, between cluster c3 and c2. If we were to merge these two clusters we end up with a new distance matrix

```
pd.DataFrame(
  [[0,0,0],
  [7,0,0],
  [5,6,0]]
, index=['c1','c4','(c2,c3)'],
columns=['c1','c4','(c2,c3)'])
#>          c1  c4  (c2,c3)
#> c1        0   0        0
#> c4        7   0        0
#> (c2,c3)   5   6        0
```

The next smallest distance is between c_1 and (c_2, c_3).

6.5 *Clustering in python*

sklearn has a cluster module with a class for Agglomerative clustering, `sklearn.cluster.AgglomerativeClustering`

```
from sklearn.cluster import AgglomerativeClustering

hc = AgglomerativeClustering(
  affinity='precomputed',
  linkage='single'
)
hc.fit(D)

## TODO improve dendrogram
#> AgglomerativeClustering(affinity='precomputed', compute_full_tree='auto',
#>                         connectivity=None, distance_threshold=None,
#>                         linkage='single', memory=None, n_clusters=2)
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram

def plot_dendrogram(model, **kwargs):
  """
  Compute the distances between each pair of children and
  a position for each child node. Then create a linkage
  matrix, and plot the dendrogram.
  """
  distance = np.arange(model.children_.shape[0])
  position = np.arange(2, model.children_.shape[0]+2)

  linkage_matrix = np.column_stack([
      model.children_, distance, position]
  ).astype(float)

  fig, ax = plt.subplots(figsize=(15, 7))

  ax = dendrogram(linkage_matrix, orientation='left', **kwargs)

  plt.tick_params(axis='x', bottom='off', top='off', labelbottom='off')
  plt.tight_layout()
  plt.show()
```

6.6 *Example: USA crime*

This dataset contains statistics, in arrests per 100,000 residents for a few crimes in each of the 50 states in 1973. There are 4 variables in total:

- Murder: Murder arrests (per 100,000)
- Assault: Assault arrests (per 100,000)
- UrbanPop: Percent urban population
- Rape: Rape arrests (per 100,000)

```
import jrpyml
arrests = jrpyml.datasets.usarrests.load_data()
```

Since we are dealing with distances between observations, we should rescale our data to ensure that measurement scale has no effect, so we can create a pipeline that will scale before performing the clustering. We can import the necessary modules:

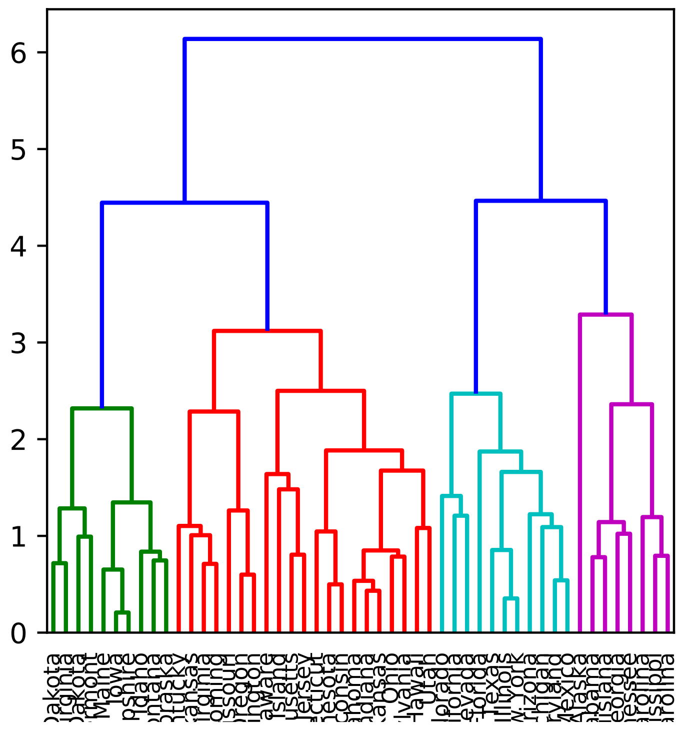
```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.cluster import AgglomerativeClustering
```

Drawing the dedrogram

Unfortunately, **sklearn** doesn't have a convenient facility to draw dendrograms, however the **scipy** library does. We can create a linkage object which will define the tree, then visualise

```
from scipy.cluster.hierarchy import linkage, dendrogram
import matplotlib.pyplot as plt

scaler = StandardScaler()
link = linkage(
  scaler.fit_transform(arrests),
  method='complete', metric='euclidean'
)
plt.figure()
dendrogram(
  link,
  leaf_label_func= lambda i: arrests.index[i]
)
#> {'icoord': [[5.0, 5.0, 15.0, 15.0], [25.0, 25.0, 35.0, 35.0], [10.0, 10.0, 30.0, 30.0], [55.0, 55
plt.show()
```



This dendrogram suggests that 4 clusters are dominant.

```
pipe = Pipeline([
  ('prep', scaler),
  ('clus', AgglomerativeClustering(
    n_clusters=4, affinity='euclidean',
    linkage='complete'
  ))
])

labels = pipe.fit_predict(arrests)
```

6.7 *K means clustering*

One of the disadvanteges of heirarchical clustering is that it is relatively computationally expensive. The number of operations scales with n^3 so when there are large numbers of observations this is often prohibitive. An alternative popular method is k-means clustering.

In K-means clustering we use Euclidean metric for calculation of distances. We need to decide a prior on a value of k to run the algorithm. With k-means we place each observation into one of the k clusters. The k-means refers to the mean vector in each cluster which are used to determine distance of each observation from the center of each cluster.

Letting x_{ij} to denote the jth observation in the the ith cluster, the ith cluster mean is simply

$$\bar{x}_i = \frac{1}{n_i}\sum_{j=1}^{n_i} x_{ij}$$

K-means clustering is in some sense similar to one way ANOVA, the key idea is reduction of variance.

We can define the total sum of squares as

$$SS_T = \sum_{i=1}^{k}\sum_{j=1}^{n_k}(x_{ij} - \bar{x})^2$$

which is independent of k. We want to split the total variation in the data set (SS_T) into two components:

- SS_W: variation within groups,
- SS_E: variation between groups,

and aim to maximise SS_E.

If

$$SS_T = SS_W + SS_E$$

where

$$SS_W = \sum_{i=1}^{k}\sum_{j=1}^{n_k}(x_{ij} - \bar{x}_i)^2$$

and $SS_E = SS_T - SS_W$,

then we maximise SS_E by minimising SS_W, effectively the variance within the clusters.

K-means algorithm

1. **Initialise**: Randomly allocate each observation to each cluster
2. **Assignment**: Assign each observation to the cluster whose mean has the least squared Euclidean distance. (closest mean)
3. **Update**: Update the new cluster means
4. **Repeat**: Return to step 2

Guaranteed to converge (to a local minimum)

6.8 Toy Example

Suppose we have the following data set

```
1, 2, 3, 10, 11, 13
```

For $k = 2$ clusters, the groups should be $\{1, 2, 3\}$ and $\{10, 11, 13\}$.

1. **Initialise**: Group 1: $\{1, 2, 13\}$ & Group 2: $\{3, 10, 11\}$. This gives group means of $\bar{x}_1 = 5.33$ and $\bar{x}_2 = 8$, with $\bar{x} = 6.67$
2. **Assignment**: Assign values to the closest mean. So we should move the value 13 to Group 2 and the value 3 to Group 1.
3. **Update**: The new groups are $\{1, 2, 3\}$ and $\{10, 11, 13\}$.
4. **Stop**: No new assignments needed.

6.9 Example: USA crime

Since we are often going to be using K-means clustering in situations that calculating a full dendrogram is prohibitive we need an alternative method for deciding on numbers of clusters. This is often done through use of an "elbow" plot. We can run the clustering algorithm for many values of k and generate a plot of the sums of squares looking for the point at which reduction of sums of squres tails off.

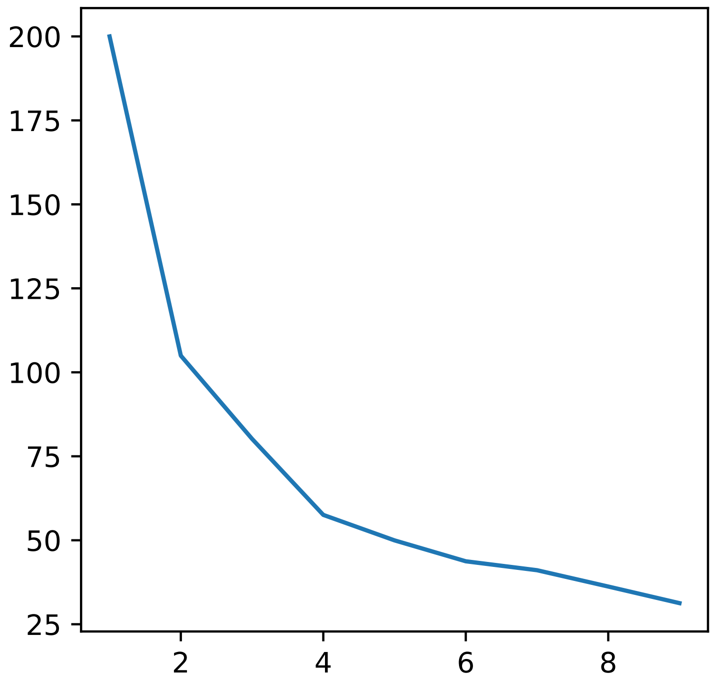
```python
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from jrpyml.datasets import usarrests

arrests = usarrests.load_data()

scaler = StandardScaler()
arrests_scaled = scaler.fit_transform(arrests)

sse = {}
estimators = {}
for k in range(1,10):
  kmeans = KMeans(n_clusters=k).fit(
    arrests_scaled
  )
  sse[k] = kmeans.inertia_
  estimators[k] = kmeans
```

```
plt.figure()
plt.plot(list(sse.keys()), list(sse.values()))
plt.show()
```



Whilst interpretation of this graph is clearly subjective, it does appear to tail off after 4 clusters, which is also what we found with the agglomerative clustering.

We can then extract the cluster labels and centers if we need them.

```
estimators[4].labels_
#> array([3, 1, 1, 3, 1, 1, 0, 0, 1, 3, 0, 2, 1, 0, 2, 0, 2, 3, 2, 1, 0, 1,
#>        2, 3, 1, 2, 2, 1, 2, 0, 1, 1, 3, 2, 0, 0, 0, 0, 0, 3, 2, 3, 1, 0,
#>        2, 0, 0, 2, 2, 0], dtype=int32)
estimators[4].cluster_centers_
#> array([[-0.49440658, -0.3864845 ,  0.58167593, -0.26431024],
#>        [ 0.70212683,  1.04999438,  0.72997363,  1.28990383],
#>        [-0.97130281, -1.11783581, -0.93954982, -0.97657842],
#>        [ 1.42622412,  0.88321132, -0.82279055,  0.01946669]])
```