8 - Predictive Models

Predictive modelling is a major area of data science for which R and Python have many tools. This section covers five major areas of popular models: Regression Models, Tree Based Models, Clustering Models, Forecasting with Time Series, and Deep Learning with Neural Networks. R and Python have various packages/modules to deal with each of these model types. There is a focus in this section on which tools to choose for each language so that one can create comparable and similar models/results. For example, the Python statsmodles module performs regression exactly the same way as R's built-in tools but choosing an alternative module such as SciPy will produce results different from R.

8.1 - Python pandas Module

The pandas module has plenty of data analysis and data structures tools. As per the pandas documentation page [19] pandas: Data Structures for Statistical Computing in Python "pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language." There is also a short introduction called "10 minutes to pandas" provided on the pandas documentation page [19] geared mainly for new users. We use the pandas module for reading in CSV files with the function `read_csv()` and also for displaying descriptive scatter plots with the function `plotting.scatter_matrix()`.
Like all of the Python modules, it can be imported with the call `import pandas` once installed. It should be noted that version 1.1.2 of pandas is being used.

```
import pandas as pd
print(pd.__version__)
```

```
1.1.2
```

8.2 - Data

The pandas function `read_csv()` is used to import the R dataset mtcars, which was previously saved to the current working directory. The 'mtcars' dataset was extracted from the 1974 Motor Trend US magazine. It comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles. The variables included are:

'mpg' - Miles/(US) gallon
'cyl' - Number of cylinders
'disp' - Displacement (cu.in.)
'hp' - Gross horsepower
'drat' - Rear axle ratio
'wt' - Weight (1000 lbs)
'qsec' - 1/4 mile time
'vs' - Engine (0 = V-shaped, 1 = straight)
'am' - Transmission (0 = automatic, 1 = manual)
'gear' - Number of forward gears
'carb' - Number of carburetors

```
cars = pd.read_csv('mtcars.csv') # import mtcars data
```

One can easily update column names in the data set. Here we update the name for the car model column.

```
# Update the 'model' column name (un-named before)
col_names = list(cars.columns)
col_names[0] = 'model'
cars.columns = col_names

cars.head() # display first 5 rows
```

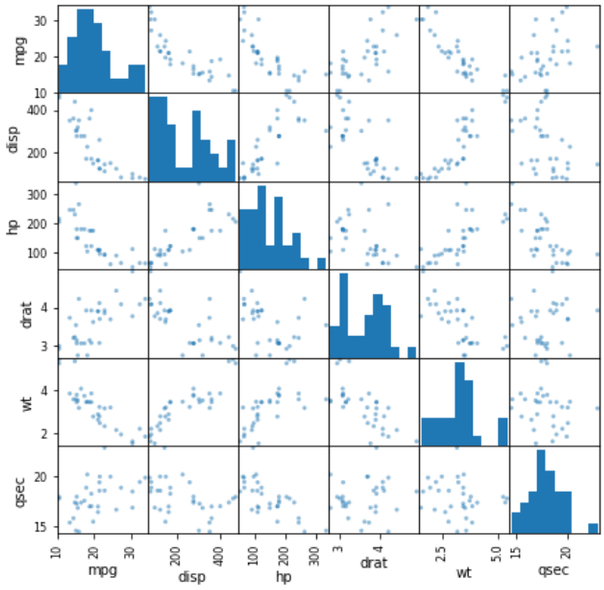
	model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

```
cars.shape # 32 rows and 12 columns
```

```
(32, 12)
```

Once the data set is saved as a pandas data frame (this is done as soon as one one reads in a file with `read_csv()`), its shape can be accessed. Here one can confirm that the full data set is of size (32, 12) hence has 32 observations each with 12 variables. One can create a comparative plot of the numeric variables in mtcars with the pandas function `plotting.scatter_matrix()`.

```
# numeric variables in mtcars
cars_numeric = cars.loc[:,['mpg','disp', 'hp', 'drat', 'wt', 'qsec']]

# display pairs plot
pd.plotting.scatter_matrix(cars_numeric)
plt.show()
```



8.2.1 - R datasets Package

The mtcars data set is downloaded directly from this package. The R datasets package is simply a package that includes various popular data sets. As per the datasets documentation page [20] The R Datasets Package "For a complete list, use `library(help = "datasets")`." Like all of the R packages it can be installed with the call `install.packages(datasets)` and imported with the call `library(datasets)`.

```
library(datasets)
```

Once the datasets package is imported (after the above call) the mtcars data set can be accessed simply with the variable `mtcars` for all R code.

8.3 - Regression

Regression is an extremely popular statistical method used to deduce relationships between a dependant variable and any number of independent variables. This section will explore simple linear regression as well as logistic regression models.

8.3.1 - Python sci-kit learn Module

Scikit-learn is a Python module for machine learning built on top of SciPy. It provides simple and efficient tools for predictive data analysis. The Scikit-learn documentation page provides great examples on classification, regression, clustering and others. To complete the regression analysis in Python the `linear_model.LinearRegression()` call is used. The documentation [22] Scikit-learn: Machine Learning in Python also notes that "from the implementation point of view, this is just plain Ordinary Least Squares (`scipy.linalg.lstsq`) or Non Negative Least Squares (`scipy.optimize.nnls`) wrapped as a predictor object." To display the regression results in a way similar to the R default, the `sklearn.metrics` module is used. This module includes score functions, performance metrics and pairwise metrics and distance computations.

The modules can be imported in the same way as all previous models using the call `import sklearn` and `from sklearn import linear_model`. It should be noted that version 0.23.2 of Scikit-learn is being used.

```
from sklearn import linear_model  # linear regression tools
import sklearn.metrics as metrics # regression output tools
print(sklearn.__version__)
```

```
0.23.2
```

8.3.2 - Python Linear Regression

Using the mtcars data, to predict the numeric variable 'qsec' (1/4 mile time) one can perform basic regression analysis. Treat 'qsec' as the dependant variable and 'mpg', 'cyl', 'disp', 'hp', 'drat', 'wt', 'vs', 'am', 'gear' and 'carb' as independent variables. This is done with the Scikit-learn functions `LinearRegression()` and `fit()`.

```
# explanatory variables (intercept included)
X = cars[['mpg',
          'cyl',
          'disp',
          'hp',
          'drat',
          'wt',
          'vs',
          'am',
          'gear',
          'carb']]

# traget or predicted variable
Y = cars[['qsec']]

# define regression model and fit
regr = linear_model.LinearRegression()
regr.fit(X, Y)
```

```
LinearRegression()
```

Model coefficients can be extracted with the call `coef_[0]`.

```
# Manually display fitted regression parameters
print('Intercept: \n', regr.intercept_[0])        # intercept
print('Coefficients: \n',
      X.columns[0], ' ', regr.coef_[0][0], '\n',  # mpg
      X.columns[1], '', regr.coef_[0][1], '\n',   # cyl
      X.columns[2], regr.coef_[0][2], '\n',       # disp
      X.columns[3], ' ', regr.coef_[0][3], '\n',  # hp
      X.columns[4], regr.coef_[0][4], '\n',       # drat
      X.columns[5], '  ', regr.coef_[0][5], '\n', # wt
      X.columns[6], '  ', regr.coef_[0][6], '\n', # vs
      X.columns[7], ' ', regr.coef_[0][7], '\n',  # am
      X.columns[8], regr.coef_[0][8], '\n',       # gear
      X.columns[9], regr.coef_[0][9], '\n',       # carb
     )
```

```
Intercept:
 17.776176929701354
Coefficients:
 mpg   0.06904768088540963
 cyl  -0.3626782334246712
 disp -0.007500837721945874
 hp   -0.0015625524378298505
 drat -0.13106359914893792
 wt    1.4963322381970534
 vs    0.970034738962025
 am   -0.9011864104662044
 gear -0.20128511286790113
 carb -0.27359751219248263
```

The fitted model which we created is the following:

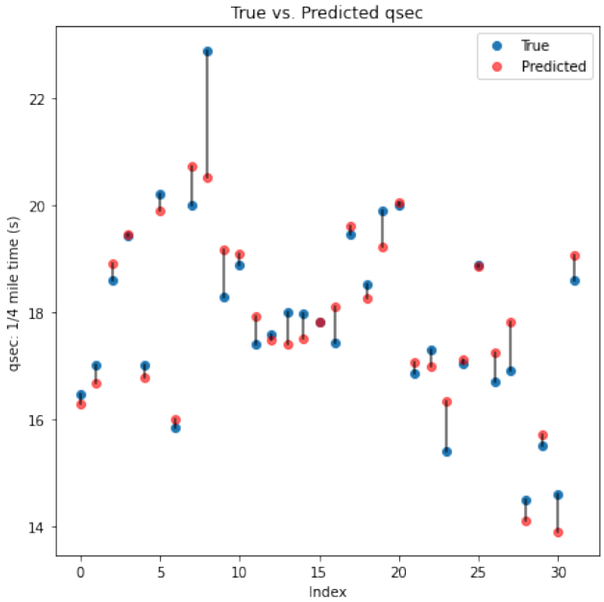
$$\hat{Y}_{qsec} = 17.78 + 0.07(X_{mpg}) - 0.36(X_{cyl}) - 0.008(X_{disp}) - 0.002(X_{hp}) - 0.13(X_{drat}) + 1.50(X_{wt}) + 0.97(X_{vs}) - 0.90(X_{am}) - 0.20(X_{gear}) - 0.27(X_{car}$$

```
# Compare predicted vs true values
results = pd.DataFrame({'qsec True': list(Y.loc[:,'qsec']),                          # true values
                        'qsec Predicted': list(regr.predict(X[list(X.columns)]).T[0])}) # predicted values
results.head()
```

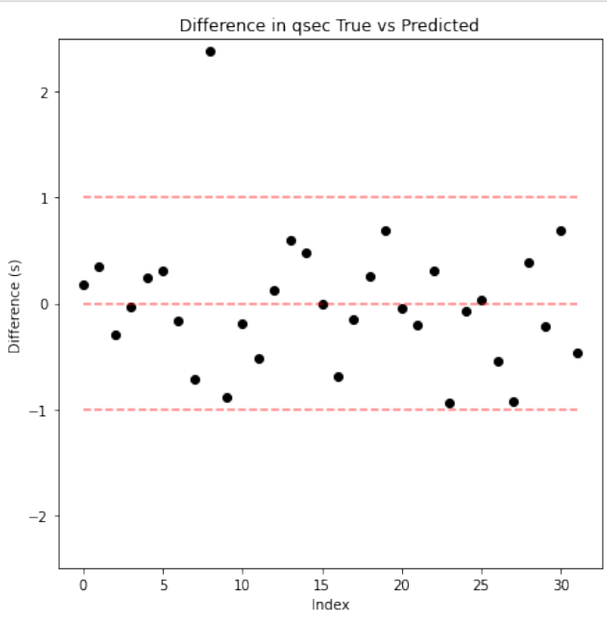
	qsec True	qsec Predicted
0	16.46	16.286620
1	17.02	16.668184
2	18.61	18.901350
3	19.44	19.470245
4	17.02	16.775677

Using the regression model one can calculate predictions for the mtcars data set and then compare those results to the true known values for 'qsec'. Plotted below is a visualization of this difference between true and predicted values.

```
# Plot of predicted vs true values
plt.plot(results[['qsec True']], # true data
         'o',                    # point type
         label = 'True')         # label for legend
plt.plot(results[['qsec Predicted']], # predicted data
         'o',
         color = "red",               # red points
         alpha = 0.6,                 # point transparency
         label = 'Predicted')
plt.plot((np.arange(32),np.arange(32)),           # line location along x axis
         (list(results.loc[:,'qsec True']),       # one end of line
          list(results.loc[:,'qsec Predicted'])), # other end of line
         c = 'black',                             # line colour
         alpha = 0.7)                             # line transparency

# Display plot
plt.xlabel('Index')
plt.ylabel('qsec: 1/4 mile time (s)')
plt.legend()
plt.title("True vs. Predicted qsec")
plt.show()
```




```
# Plot the differences between true and predicted:
y_true = list(results.loc[:,'qsec True'])        # true
y_pred = list(results.loc[:,'qsec Predicted'])   # predicted
difference = np.array(y_true) - np.array(y_pred) # find differences

# Plot differences
plt.plot(difference,
         "o",
         color = 'black')

# Plot lines (N(0,1) has mean 0 and standard deviation 1)
plt.plot(np.arange(32),    # line at y = 0
         np.zeros(32),
         color = 'red',
         alpha = 0.5,
         linestyle = "--")
plt.plot(np.arange(32),   # line at y = 1
         np.zeros(32) + 1,
         color = 'red',
         alpha = 0.5,
         linestyle = "--")
plt.plot(np.arange(32),    # line at y = -1
         np.zeros(32) - 1,
         color = 'red',
         alpha = 0.5,
         linestyle = "--")

# Display plot
plt.ylim([-2.5, 2.5])
plt.xlabel("Index")
plt.ylabel('Difference (s)')
plt.title('Difference in qsec True vs Predicted')
plt.show()
```



Regression makes the assumptions that errors should be normally distributed as $N(0,1)$.

```
# Calculate some regression metrics
mae = metrics.mean_absolute_error(y_true, y_pred)     # absolute error
mse = metrics.mean_squared_error(y_true, y_pred)      # squared error
msle = metrics.mean_squared_log_error(y_true, y_pred) # squared log error
r2 = metrics.r2_score(y_true, y_pred)                 # r-squared
```

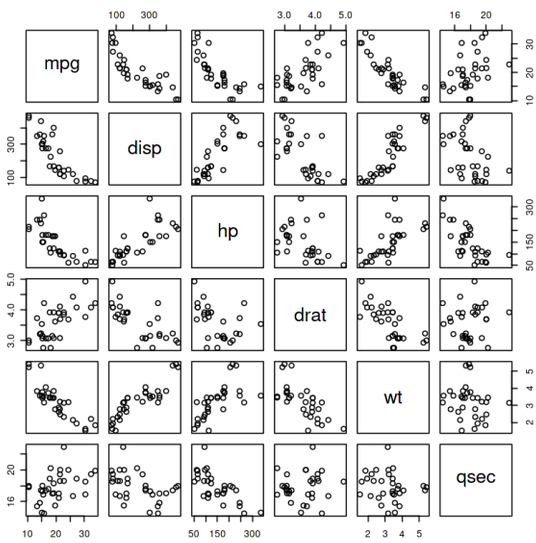
```r
# Display metrics
print('r2: ', round(r2,5))
print('Adjusted r2: ', round(1 - (1-r2)*(32-1)/(32-10-1),5)) # adjusted r-squared
print('Mean Absolute Error: ', round(mae,5))
print('Mean Squared Error: ', round(mse,5))
print('Mean Squared log Error: ', round(msle,5))
```

```
r2:  0.87469
Adjusted r2:  0.81502
Mean Absolute Error:  0.44006
Mean Squared Error:  0.38762
Mean Squared log Error:  0.00097
```

8.3.3 - R Linear Regression

Working with the same mtcars dataset one can get the exact same results in R as with Python. R has a built-in function `pairs()` which produces a comparative plot of the numeric mtcars variables, similar to the pandas function `plotting.scatter_matrix()` in Python.

```r
pairs(mtcars[c('mpg', 'disp', 'hp', 'drat', 'wt', 'qsec')]) # pairs plot with numeric variables
```



Using the built-in R function `lm()` (linear model) one can easily define a regression model using the formula notation "y ~ x1 + x2 + ... + xn". It should be noted that making use of the dot "~ ." in the formula `qsec ~ .` will perform regression on qsec with all other variables in mtcars. One can then use the R `summary()` function to produce regression results.

```r
lreg <- lm(qsec ~ ., data = mtcars) # linear regression model
summary(lreg)                       # model summary
```

```
Call:
lm(formula = qsec ~ ., data = mtcars)

Residuals:
     Min       1Q   Median       3Q      Max 
-0.93377 -0.33421 -0.03696  0.31389  2.38743 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept) 17.776177   3.875998   4.586  0.00016 ***
mpg          0.069048   0.061462   1.123  0.27394    
cyl         -0.362678   0.292621  -1.239  0.22887    
disp        -0.007501   0.004985  -1.505  0.14730    
hp          -0.001563   0.006449  -0.242  0.81089    
drat        -0.131064   0.476002  -0.275  0.78574    
wt           1.496332   0.500469   2.990  0.00698 ** 
vs           0.970035   0.572767   1.694  0.10512    
am          -0.901186   0.585218  -1.540  0.13851    
gear        -0.201285   0.432798  -0.465  0.64666    
carb        -0.273598   0.233143  -1.174  0.25373    
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.7685 on 21 degrees of freedom
Multiple R-squared:  0.8747,	Adjusted R-squared:  0.815 
F-statistic: 14.66 on 10 and 21 DF,  p-value: 2.438e-07
```

The models fitted values can be extracted with the R function `fitted.values()`.

```r
# Get predicted values
pred <- fitted.values(lreg)

# Display results
results <- cbind(mtcars$qsec, pred)
colnames(results) <- c('qsec True', 'qsec Predicted')
head(results, 5)
```

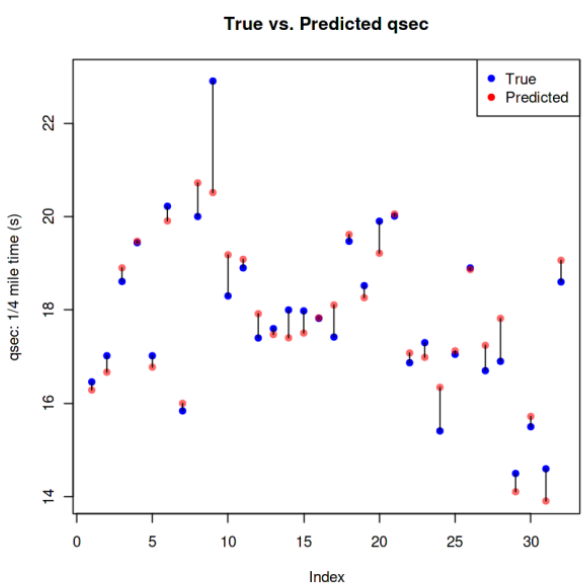
A matrix: 5 x 2 of type dbl

	qsec True	qsec Predicted
Mazda RX4	16.46	16.28662

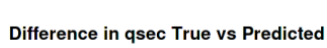
Mazda RX4 Wag	17.02	16.66818
Datsun 710	18.61	18.90135
Hornet 4 Drive	19.44	19.47025
Hornet Sportabout	17.02	16.77568

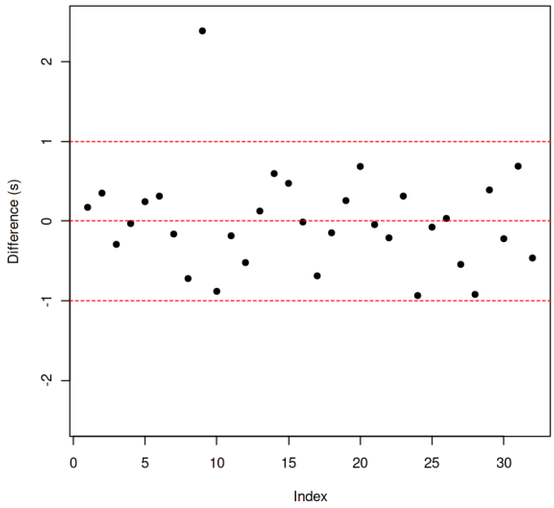
One can observe that the coefficients, predicted values, and model summary in R are the same as in Python.

```
# Plot of predicted vs true values
plot(mtcars$qsec,                        # true data
     col = 'blue',                       # true points blue
     pch = 16,                           # filled circle points
     ylab = 'qsec: 1/4 mile time (s)', # y axis label
     xlab = 'Index',
     main = "True vs. Predicted qsec",
     ylim = c(14, 23))

# Add predicted points
points(1:32,
       pred,                                       # predicted data
       col = adjustcolor('red', alpha.f = 0.6), # transparency with red points
       pch = 16)

# Add lines
for (i in 1:32) {
    lines(c(i,i), c(mtcars$qsec[i], pred[i])) # between predicted vs true points
}

# Add legend
legend('topright' ,
       legend = c("True", "Predicted"),
       pch = 16,
       col = c("Blue", "red"))
```




```
# Calculate the difference between true and predicted values
difference <- mtcars$qsec - pred

# Plot differences
plot(difference,  # data
     col = 'black',
     pch = 16,
     xlab = "Index",
     ylab = 'Difference (s)',
     main = 'Difference in qsec True vs Predicted',
     ylim = c(-2.5, 2.5))

# Add lines (N(0,1) has mean 0 and standard deviation 1)
abline(h = c(0, 1, -1), # location along y axis for horizontal lines
       col = 'red',     # red colour
       lty = 2)         # line width
```






```
# Calculate some regression metrics
y_true <- mtcars$qsec
y_pred <- pred
mae <- mean(abs(difference))                        # absolute error
mse <- mean(difference^2)                           # squared error
msle <- mean((log(y_true + 1) - log(y_pred + 1))^2) # squared log error
r2 <- 1 - mse/mean((y_true - mean(y_true))^2)       # r-squared
r2_adj <- 1 - (1-r2)*(32-1)/(32-10-1)               # adjusted r-squared

# Display metrics
print(paste('r2: ', round(r2, 5)))
print(paste('Adjusted r2: ', round(r2_adj, 5)))
print(paste('Mean Absolute Error: ', round(mae, 5)))
print(paste('Mean Squared Error: ', round(mse, 5)))
print(paste('Mean Squared log Error: ', round(msle, 5)))
```

```
[1] "r2:  0.87469"
[1] "Adjusted r2:  0.81502"
[1] "Mean Absolute Error:  0.44006"
[1] "Mean Squared Error:  0.38762"
[1] "Mean Squared log Error:  0.00097"
```

All metrics again are the same as when calculated using Python.

8.3.4 - Python Logistic Regression

The statsmodels module is being used for logistic regression here since the results will be identical to R's logistic regression tool `glm()`. Performing logistic regression using the Scikit-learn module will produce different results compared to R. More information on why this is the case can be found on the statsmodels documentation page [11]. Here one can find details on the logistic regression methods used and why they are indeed equivalent to R's.

```
import statsmodels.api as sm
print(sm.__version__)
```

```
0.13.2
```

Looking to predict the classification variable 'am' (automatic or manual transmission) one can perform basic logistic regression analysis. In order for the logistic regression model fitting to mimic that done in R, one must use the statsmodels package. This package includes the `Logit()` function from which one can define the logistic model and the `fit()` function which will fit the model in the same way as R. One will also notice how the statsmodels package includes the function `summary()` which produces summary results similar to R's `summary()` function.

```
# explanatory variables
X = sm.add_constant(cars[['mpg','hp','qsec']])

# target or predicted variable
Y = cars[['am']]

# define regression model and fit
log_regr = sm.Logit(Y, X)
result = log_regr.fit()

# display model summary
result.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.116878
         Iterations 11
```

Logit Regression Results

Dep. Variable:	am	**No. Observations:**	32
Model:	Logit	**Df Residuals:**	28
Method:	MLE	**Df Model:**	3
Date:	Mon, 08 Aug 2022	**Pseudo R-squ.:**	0.8270
Time:	12:01:27	**Log-Likelihood:**	-3.7401
converged:	True	**LL-Null:**	-21.615
Covariance Type:	nonrobust	**LLR p-value:**	8.459e-08

	coef	std err	z	P>|z|	[0.025	0.975]

const	23.7130	44.076	0.538	0.591	-62.674	110.100
mpg	2.2964	1.624	1.414	0.157	-0.886	5.479
hp	0.0073	0.047	0.155	0.877	-0.085	0.100
qsec	-4.0410	2.891	-1.398	0.162	-9.708	1.626

Possibly complete quasi-separation: A fraction 0.44 of observations can be perfectly predicted. This might indicate that there is complete quasi-separation. In this case some parameters will not be identified.

The fitted model which we created is the following:

$$\log(\frac{\hat{Y}_{am}}{1-\hat{Y}_{am}}) = 23.71 + 2.30(X_{mpg}) + 0.007(X_{hp}) - 4.04(X_{qsec})$$
$$\Rightarrow \hat{Y}_{am} = \frac{e^{(23.71+2.30(X_{mpg})+0.007(X_{hp})-4.04(X_{qsec}))}}{1+e^{(23.71+2.30(X_{mpg})+0.007(X_{hp})-4.04(X_{qsec}))}}$$

```
pred = list(round(result.predict(X), 0)) # calculate predicted values
pred = [int(i) for i in pred]            # round probabilities to correct {0,1} classification group

# Display results
results = pd.DataFrame({'am True': list(Y.loc[:,'am']), # true
                        'am Predicted': pred})          # predicted
results
```

	am True	am Predicted
0	1	1
1	1	1
2	1	1
3	0	0
4	0	0
5	0	0
6	0	0
7	0	0
8	0	0
9	0	0
10	0	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	1	1
18	1	1
19	1	1
20	0	0
21	0	0
22	0	0
23	0	0
24	0	1
25	1	1
26	1	1
27	1	1
28	1	1
29	1	1
30	1	1
31	1	0

```
# Calculate some accuracy metrics
cm = metrics.confusion_matrix(list(Y.loc[:,'am']), pred)    # confusion matrix
tot_acc = metrics.accuracy_score(list(Y.loc[:,'am']), pred) # total accuracy
manual_acc = cm[1,1]/sum(cm[1,])                            # manual group accuracy
auto_acc = cm[0,0]/sum(cm[0,])                              # automatic group accuracy
```

```r
# Display metrics
print('Total Accuracy: ', tot_acc)
print('Manual (am = 1) Accuracy: ', round(manual_acc,4))
print('Automatic (am = 0) Accuracy: ', round(auto_acc, 4))
```

```
Total Accuracy:  0.9375
Manual (am = 1) Accuracy:  0.9231
Automatic (am = 0) Accuracy:  0.9474
```

8.3.5 - R Logistic Regression

The logistic regression model in R can be defined wiht the `glm()` (general linear model) function and the parameter `family = binomial(link='logit')` to enforce the logistic form. The formula notation is used here where the independent variables are manually inserted with the call `am ~ mpg + hp + qsec`.

```r
# Logistic regression
mtcars$am <- as.factor(mtcars$am) # update 'am' variable to act as a factor variable

# Build model
logreg <- glm(am ~ mpg + hp + qsec, data = mtcars, family = binomial(link='logit'))

# Model summary
summary(logreg)
```

```
Call:
glm(formula = am ~ mpg + hp + qsec, family = binomial(link = "logit"),
    data = mtcars)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-1.25597  -0.04564  -0.00055   0.00211   1.84909

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) 23.712985  44.069519   0.538    0.591
mpg          2.296430   1.623232   1.415    0.157
hp           0.007295   0.047199   0.155    0.877
qsec        -4.040952   2.890460  -1.398    0.162

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 43.2297  on 31  degrees of freedom
Residual deviance:  7.4802  on 28  degrees of freedom
AIC: 15.48

Number of Fisher Scoring iterations: 9
```

One can observe how the model summary and coefficients are the same as the Python version.

```r
# Compare true vs predicted values
true <- mtcars$am                        # true
pred <- round(fitted.values(logreg), 0) # predicted

# Display results
results <- cbind(true, pred + 1)
colnames(results) <- c('am True', 'am Predicted')
results
```

A matrix: 32 x 2 of type dbl

	am True	am Predicted
Mazda RX4	2	2
Mazda RX4 Wag	2	2
Datsun 710	2	2
Hornet 4 Drive	1	1
Hornet Sportabout	1	1
Valiant	1	1
Duster 360	1	1
Merc 240D	1	1
Merc 230	1	1
Merc 280	1	1
Merc 280C	1	1
Merc 450SE	1	1
Merc 450SL	1	1
Merc 450SLC	1	1
Cadillac Fleetwood	1	1
Lincoln Continental	1	1
Chrysler Imperial	1	1
Fiat 128	2	2
Honda Civic	2	2
Toyota Corolla	2	2
Toyota Corona	1	1
Dodge Challenger	1	1

AMC Javelin	1	1
Camaro Z28	1	1
Pontiac Firebird	1	2
Fiat X1-9	2	2
Porsche 914-2	2	2
Lotus Europa	2	2
Ford Pantera L	2	2
Ferrari Dino	2	2
Maserati Bora	2	2
Volvo 142E	2	1

```
# Calculate some accuracy metrics
tot_acc <- sum(pred == true)/length(pred)                        # total accuracy
index1 <- which(true == 1)                                       # indices of manual group
n1 <- length(index1)                                             # size of manual group
manual_acc <- sum(true[index1] == pred[index1])/n1               # manual group accuracy
auto_acc <- sum(true[-index1] == pred[-index1])/(length(pred)-n1) # automatic group accuracy

# Display metrics
print(paste('Total Accuracy: ', tot_acc))
print(paste('Manual (am = 1) Accuracy: ', round(manual_acc, 4)))
print(paste('Automatic (am = 0) Accuracy: ', round(auto_acc, 4)))
```

```
[1] "Total Accuracy:  0.9375"
[1] "Manual (am = 1) Accuracy:  0.9231"
[1] "Automatic (am = 0) Accuracy:  0.9474"
```

Clearly one can see how we achieve the same predictions and hence identical accuracy scores.

8.4 - Decision Trees

Decision trees are a type of supervised learning model that can be used for both regression and classification. The Python Scikit-learn modules documentation page [22] has a great section dedicated to decision trees titled "1.10. Decision Trees".

8.4.1 - Regression Trees with scikit-learn

The sklearn.tree module has great tools for classification and regression problems. The decision tree documentation page on scikit-lean is extremely rich with great content for further reading. The tree documentation page [22] also makes the note that "The default values for the parameters controlling the size of the trees (e.g. max_depth, min_samples_leaf, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values."

```
from sklearn import tree
```

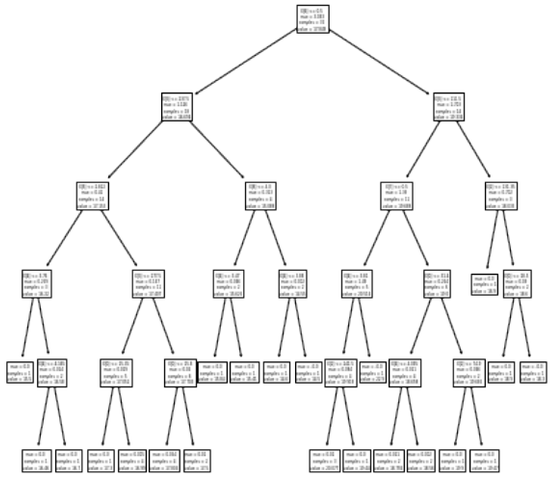
From sklearn one can use the function `DecisionTreeRegressor()` and `fit()` to fit a regression decision tree.

```
# explanatory variables
X = cars[['mpg',
          'cyl',
          'disp',
          'hp',
          'drat',
          'wt',
          'vs',
          'am',
          'gear',
          'carb']]

# traget or predicted variable
Y = cars[['qsec']]

# model
dt = tree.DecisionTreeRegressor(max_depth = 5)
result = dt.fit(X, Y)
```

There also exist functions for plotting a visualization of a built decision tree model. This can be done with the function `plot_tree()`. It should be noted that very large trees with many leaves will be difficult to view.

```
tree.plot_tree(dt)
plt.title('qsec Decsion Tree in Python')
plt.show()
```

qsec Decsion Tree in Python

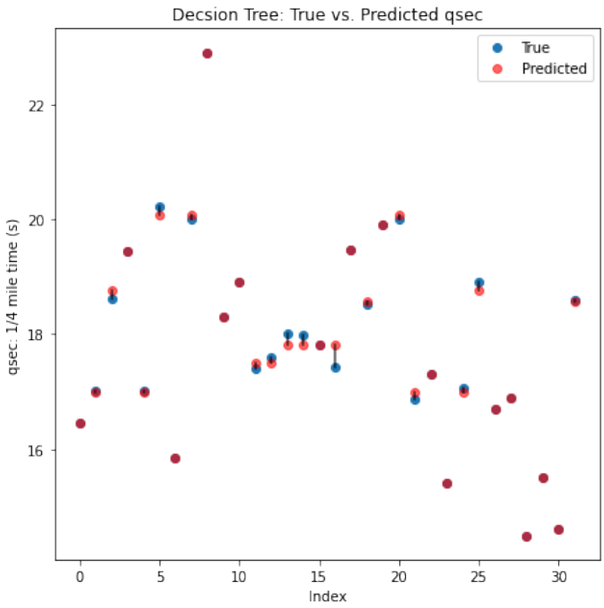


```
pred = result.predict(X)
compare = pd.DataFrame({'qsec True': list(Y.loc[:,'qsec']), # true
                        'qsec Predicted': pred})            # predicted
compare.head()
```

	qsec True	qsec Predicted
0	16.46	16.460
1	17.02	16.990
2	18.61	18.755
3	19.44	19.440
4	17.02	16.990

Using the same `predict()` function one can gather predicted values for the decision tree model and compare those to the true values for the 'qsec' variable.

```
# Plot of predicted vs true values
plt.plot(Y,               # true data
         'o',             # point type
         label = 'True') # label for legend
plt.plot(pred,                # predicted data
         'o',
         color = "red",       # red points
         alpha = 0.6,         # point transparency
         label = 'Predicted')
plt.plot((np.arange(32), np.arange(32)), # line location along x axis
         (list(Y.loc[:,'qsec']), pred),
         c = 'black',                    # line colour
         alpha = 0.7)                    # line transparency

# Display plot
plt.xlabel('Index')
plt.ylabel('qsec: 1/4 mile time (s)')
plt.legend()
plt.title("Decsion Tree: True vs. Predicted qsec")
plt.show()
```



One can notice the higher level of accuracy compared to the regression model. It should be noted that using a smaller maximum depth (for example `max_depth = 2`) will result in less accurate predictions.

```
y_true = list(compare.loc[:,'qsec True'])        # true
y_pred = list(compare.loc[:,'qsec Predicted'])   # predicted
```

```
# Calculate some regression metrics
mae = metrics.mean_absolute_error(y_true, y_pred)     # absolute error
mse = metrics.mean_squared_error(y_true, y_pred)      # squared error
msle = metrics.mean_squared_log_error(y_true, y_pred) # squared log error

# Display metrics
print('Mean Absolute Error: ', round(mae,5))
print('Mean Squared Error: ', round(mse,5))
print('Mean Squared log Error: ', round(msle,5))
```

```
Mean Absolute Error:  0.05833
Mean Squared Error:  0.01041
Mean Squared log Error:  3e-05
```

8.4.2 - R rpart Package

One can make use of the rpart package in R to create decision trees. As per the rpart documentation page [25] rpart: Recursive Partitioning and Regression Trees, the package implements "Recursive partitioning for classification, regression and survival trees". Decision tree models have many hyper-parameters, so in order to have the rpart functions behave similarly to the Python sklearn tree functions one must adjust the hyper-parameters so that they are the same. Within rpart one must set the 'minsplit', 'minbucket', 'maxdepth', and 'cp' to the default values that sklearn uses. It should be noted that the resulting trees still may have a different structure but their predictive output should be almost identical.

As with other R packages, rpart can be installed with the call `library(rpart)`. Also the `rpart.plot` functions are imported so one can use the tree visualization tools. It should be noted that version 4.1.15 of rpart is being used.

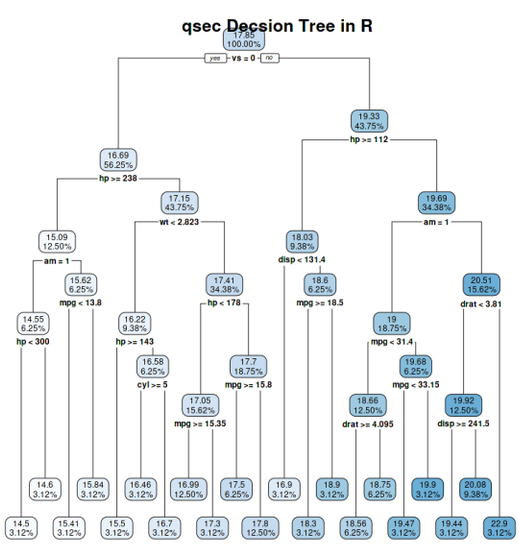
```
library(rpart)      # decsion tree modelling
library(rpart.plot) # decsion tree visualizations
packageVersion("rpart")
```

```
[1] ‘4.1.15’
```

8.4.3 - R Regression Trees

One can use the rpart function `rpart` in the same way as `lm()` or `glm()` providing a formula and control parameters along with the data set.

```
# Decsion tree model
dt <- rpart(qsec ~ .,                              # All variables included
            data = mtcars,                         # data
            control = rpart.control(minsplit = 2,  # minimum observations for node split
                                    minbucket = 1, # minimum observations in leaf node
                                    maxdepth = 5,  # maximum depth of any node in tree
                                    cp = 0))       # complexity parameter for pruning

# Plot tree strcuture
rpart.plot(dt, digits = 4)
title("qsec Decsion Tree in R")
```

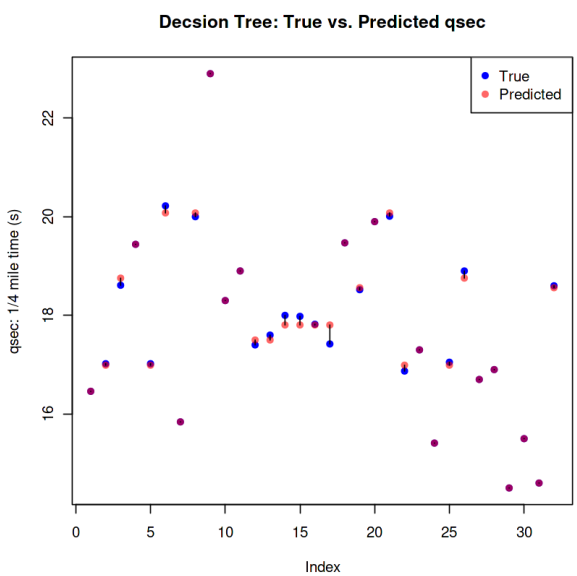



```
# Calculate predicted values
pred <- predict(dt, newdata = mtcars)
true <- mtcars$qsec

# Display results
results <- cbind(true, pred)
colnames(results) <- c('qsec True', 'qsec Predicted')
head(results, 5)
```

A matrix: 5 x 2 of type dbl

	qsec True	qsec Predicted
Mazda RX4	16.46	16.460
Mazda RX4 Wag	17.02	16.990
Datsun 710	18.61	18.755
Hornet 4 Drive	19.44	19.440
Hornet Sportabout	17.02	16.990

```r
# Plot of true values
plot(mtcars$qsec,
     col = 'blue',
     pch = 16,
     ylab = 'qsec: 1/4 mile time (s)',
     xlab = 'Index',
     main = "Decsion Tree: True vs. Predicted qsec")

# Add predicted points
points(1:32,
       pred,
       col = adjustcolor('red', 0.6),
       pch = 16)

# Add lines between predicted vs true points
for (i in 1:32) {
  lines(c(i,i), c(mtcars$qsec[i], pred[i]))
}

# Add legend
legend('topright' ,
       legend = c("True", "Predicted"),
       pch = 16,
       col = c("Blue",
               adjustcolor("red", 0.6)))
```




```r
# Calculate some regression metrics
difference <- mtcars$qsec - pred                # differences
mae <- mean(abs(difference))                    # absolute error
mse <- mean(difference^2)                       # squared error
msle <- mean((log(true + 1) - log(pred + 1))^2) # squared log error

# Display metrics
print(paste('Mean Absolute Error: ', round(mae, 5)))
print(paste('Mean Squared Error: ', round(mse, 5)))
print(paste('Mean Squared log Error: ', round(msle, 5)))
```

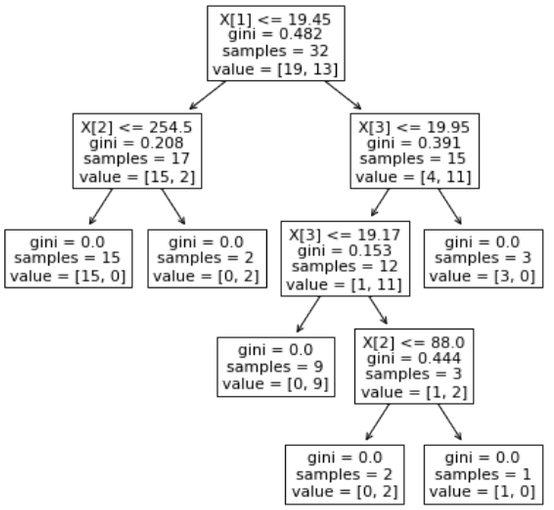
```
[1] "Mean Absolute Error:  0.05833"
[1] "Mean Squared Error:  0.01041"
[1] "Mean Squared log Error:  3e-05"
```

8.4.4 - Python Classification Trees

From sklearn one can use the function `DecisionTreeClassifier()` and `fit()` to fit a classification decision tree.

```python
X = sm.add_constant(cars[['mpg','hp','qsec']]) # explanatory variables
Y = cars[['am']]                               # target or predicted variable

# Build dt model with maximum depth equal to 4
class_dt = tree.DecisionTreeClassifier(max_depth = 4)

# Fit dt model
class_result = class_dt.fit(X, Y)

# Plot tree strcuture
tree.plot_tree(class_result)
plt.title('am Decsion Tree in Python')
plt.show()
```

am Decsion Tree in Python



Note that using a smaller `max_depth` parameters will result in a "shorter" tree which is easier to display.

```
# Calculate predictde values
pred = class_result.predict(X)

# Comapre predicted vs true values
compare = pd.DataFrame({'am True': list(Y.loc[:,'am']), # true
                        'am Predicted': pred})          # predicted
compare
```

	am True	am Predicted
0	1	1
1	1	1
2	1	1
3	0	0
4	0	0
5	0	0
6	0	0
7	0	0
8	0	0
9	0	0
10	0	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	1	1
18	1	1
19	1	1
20	0	0
21	0	0
22	0	0
23	0	0
24	0	0
25	1	1
26	1	1
27	1	1
28	1	1
29	1	1
30	1	1
31	1	1

```
# Calculate some accuracy metrics
cm = metrics.confusion_matrix(list(Y.loc[:,'am']), pred)    # confusion matrix
tot_acc = metrics.accuracy_score(list(Y.loc[:,'am']), pred) # total accuracy
manual_acc = cm[1,1]/sum(cm[1,])                            # manual group accuracy
auto_acc = cm[0,0]/sum(cm[0,])                              # automatic group accuracy
```

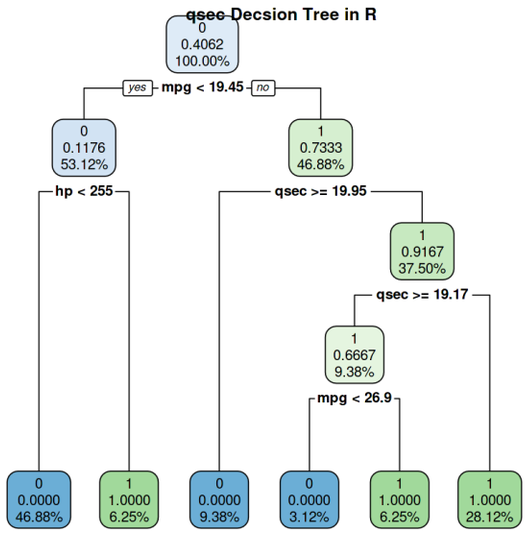
```
# Display metrics
print('Total Accuracy: ', tot_acc)
print('Manual (am = 1) Accuracy: ', round(manual_acc,4))
print('Automatic (am = 0) Accuracy: ', round(auto_acc, 4))
```

```
Total Accuracy:  1.0
Manual (am = 1) Accuracy:  1.0
Automatic (am = 0) Accuracy:  1.0
```

8.4.5 - R Classification Trees

Using the rpart package one can create a decision tree model using the function `rpart()` with formula notation to specify the model, the same as with R's `glm()` and `lm()` functions.

```
# Decsion tree model
mtcars$am <- as.factor(mtcars$am) # update 'am' variable to act as a factor variable
class_dt <- rpart(am ~ mpg + hp + qsec,
                  data = mtcars,                          # data
                  control = rpart.control(minsplit = 2,  # minimum observations for node split
                                          minbucket = 1, # minimum observations in leaf node
                                          maxdepth = 5,  # maximum depth of any node in tree
                                          cp = 0))       # complexity parameter for pruning

# Plot tree strcuture
rpart.plot(class_dt, digits = 4)
title("qsec Decsion Tree in R")
```




```
# Calculate predicted values
pred <- predict(class_dt, newdata = mtcars, type = 'class')

# Display results
results <- cbind(mtcars$am, pred)
colnames(results) <- c('am True', 'am Predicted')
results
```

A matrix: 32 x 2 of type int

	am True	am Predicted
Mazda RX4	2	2
Mazda RX4 Wag	2	2
Datsun 710	2	2
Hornet 4 Drive	1	1
Hornet Sportabout	1	1
Valiant	1	1
Duster 360	1	1
Merc 240D	1	1
Merc 230	1	1
Merc 280	1	1
Merc 280C	1	1
Merc 450SE	1	1
Merc 450SL	1	1
Merc 450SLC	1	1
Cadillac Fleetwood	1	1
Lincoln Continental	1	1
Chrysler Imperial	1	1
Fiat 128	2	2
Honda Civic	2	2
Toyota Corolla	2	2

Toyota Corona	1	1
Dodge Challenger	1	1
AMC Javelin	1	1
Camaro Z28	1	1
Pontiac Firebird	1	1
Fiat X1-9	2	2
Porsche 914-2	2	2
Lotus Europa	2	2
Ford Pantera L	2	2
Ferrari Dino	2	2
Maserati Bora	2	2
Volvo 142E	2	2

```
true <- mtcars$am
# Calculate some accuracy metrics
tot_acc <- sum(pred == true)/length(pred)                        # total accuracy
index1 <- which(true == 1)                                       # indices of manual group
n1 <- length(index1)                                             # size of manual group
manual_acc <- sum(true[index1] == pred[index1])/n1               # manual group accuracy
auto_acc <- sum(true[-index1] == pred[-index1])/(length(pred)-n1) # automatic group accuracy

# Display metrics
print(paste('Total Accuracy: ', tot_acc))
print(paste('Manual (am = 1) Accuracy: ', round(manual_acc, 4)))
print(paste('Automatic (am = 0) Accuracy: ', round(auto_acc, 4)))
```

```
[1] "Total Accuracy:  1"
[1] "Manual (am = 1) Accuracy:  1"
[1] "Automatic (am = 0) Accuracy:  1"
```

8.5 - Clustering

Clustering models are unsupervised learning models which seek to separate data into a given number of groups such that the data points in each of the groups are "most similar" to one another. There are different methods for determining data point similarity but here the k-means method is exclusively used.

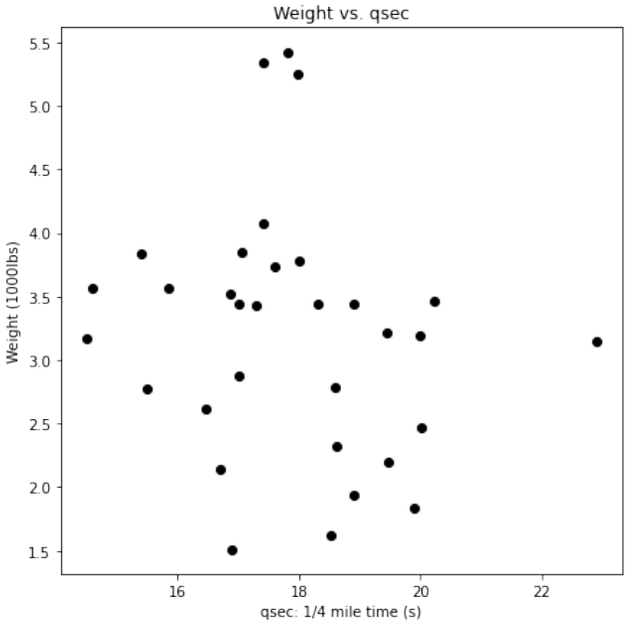
8.5.1 - Python Clustering

The Python module sklearn has a k-means clustering method which can be imported with the call `from sklearn.cluster import KMeans`. As per the sklearn documentation page [22] Scikit-learn: Machine Learning in Python "The k-means problem is solved using either Lloyd's or Elkan's algorithm. In practice, the k-means algorithm is very fast (one of the fastest clustering algorithms available), but it falls in local minima. That's why it can be useful to restart it several times."

```
from sklearn.cluster import KMeans # kmeans clustering tools
```

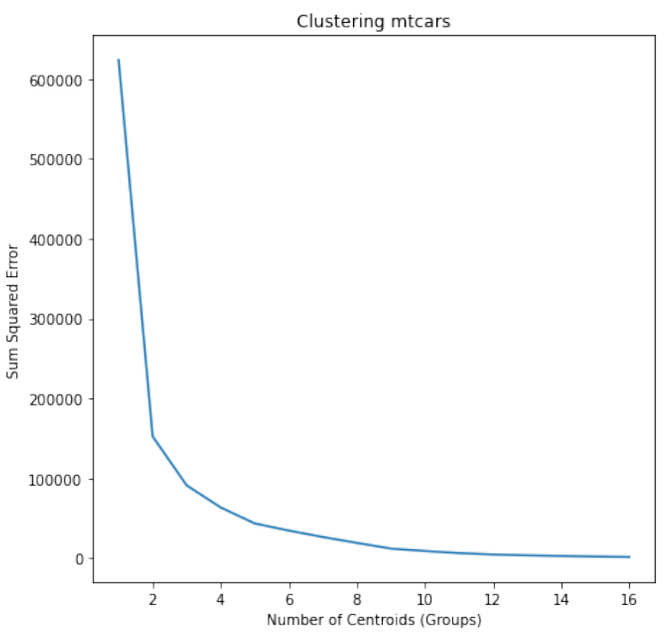
To better visualize the clustering result one can plot the data points along the weight vs qsec axis in order to get a Cartesian plane. It should be noted that any two axes can be chosen to display the data points; the clustering groups will not change.

```
# remove model column
data = cars[['mpg',
             'cyl',
             'disp',
             'hp',
             'drat',
             'wt',
             'vs',
             'am',
             'gear',
             'carb',
             'qsec']]

# Plot original data in the 'weight'-'qsec space'
plt.plot(data[['qsec']], # qsec
         data[['wt']],   # weight
         "o",
         color = 'black')
plt.xlabel('qsec: 1/4 mile time (s)')
plt.ylabel('Weight (1000lbs)')
plt.title('Weight vs. qsec')
plt.show()
```



Fifteen different clustering models are built with the number of groups in each model being equal to 2, 3, 4, ..., 15, and 16. One can expect the k-means error calculated to decrease as the number of groups increases. Obviously setting the number of groups equal to the number of data points will result in an error of 0 but that essentially defeats the purpose of clustering.

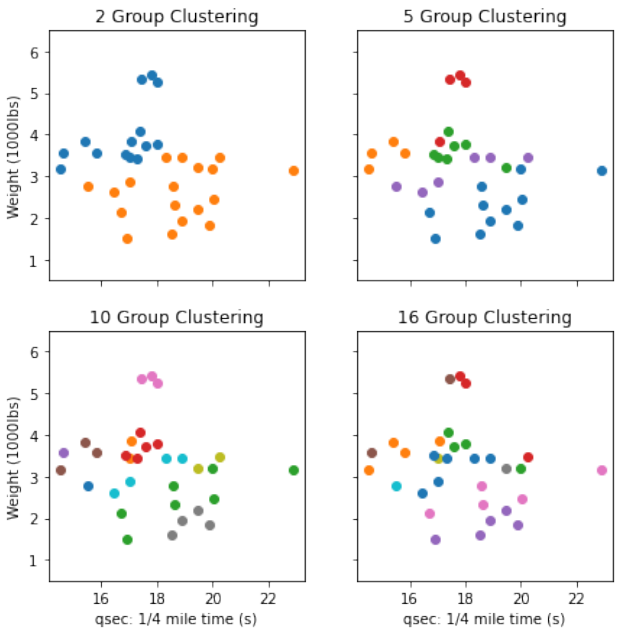
```
# Track sum of squares error for each clustering model
errors = []
centroids = list(range(1,17)) # testing number of groups equal to 1,2,...,16

# Build models and record error
for k in centroids:
    clust = KMeans(n_clusters = k).fit(data)
    errors.append(clust.inertia_)

# Display results
plt.plot(centroids, errors)
plt.xlabel('Number of Centroids (Groups)')
plt.ylabel('Sum Squared Error')
plt.title('Clustering mtcars')
plt.show()
```




```
# Build models for visual comparison
kmc2 = KMeans(n_clusters = 2).fit(data) # 2 groups
pred2 = kmc2.predict(data)

kmc5 = KMeans(n_clusters = 5).fit(data) # 5 groups
pred5 = kmc5.predict(data)

kmc10 = KMeans(n_clusters = 10).fit(data) # 10 groups
pred10 = kmc10.predict(data)

kmc16 = KMeans(n_clusters = 16).fit(data) # 16 groups
pred16 = kmc16.predict(data)
```

```
# Display the clustering results in the 'weight'-'qsec' space:
```

```
# 2 group model
plt.subplot(2,2,1)
for i in np.unique(pred2):
    plt.scatter(data.loc[pred2 == i][['qsec']], # qsec by group
                data.loc[pred2 == i][['wt']])   # weight by group
plt.ylabel('Weight (1000lbs)')
plt.title('2 Group Clustering')
plt.xticks(color = 'w')
plt.ylim([0.5, 6.5])

# 5 group model
plt.subplot(2,2,2)
for i in np.unique(pred5):
    plt.scatter(data.loc[pred5 == i][['qsec']],
                data.loc[pred5 == i][['wt']])
plt.title('5 Group Clustering')
plt.xticks(color = 'w')
plt.yticks(color = 'w')
plt.ylim([0.5, 6.5])

# 10 group model
plt.subplot(2,2,3)
for i in np.unique(pred10):
    plt.scatter(data.loc[pred10 == i][['qsec']],
                data.loc[pred10 == i][['wt']])
plt.xlabel('qsec: 1/4 mile time (s)')
plt.ylabel('Weight (1000lbs)')
plt.title('10 Group Clustering')
plt.ylim([0.5, 6.5])

# 16 group model
plt.subplot(2,2,4)
for i in np.unique(pred16):
    plt.scatter(data.loc[pred16 == i][['qsec']],
                data.loc[pred16 == i][['wt']])
plt.xlabel('qsec: 1/4 mile time (s)')
plt.title('16 Group Clustering')
plt.yticks(color = 'w')
plt.ylim([0.5, 6.5])
plt.show()
```

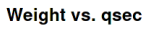


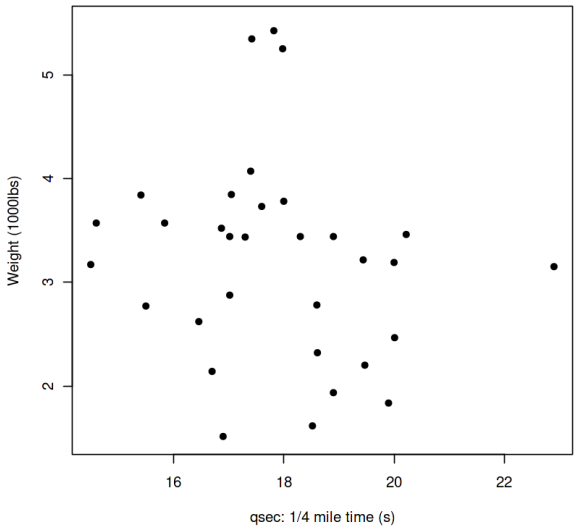
Four of the model results are displayed above where the number of groups or clusters is set to 2, 5, 10 and 16.

8.5.2 - R Clustering

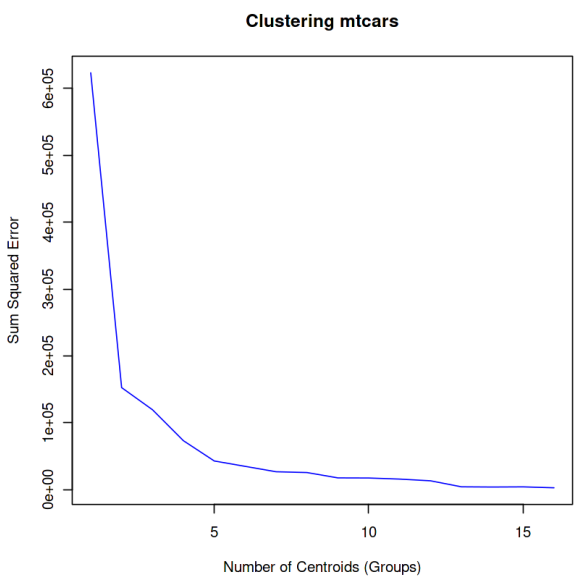
Clustering in R can be performed in a way very similar to how the sklearn tools are used. R has a built-in function `kmeans()` that is used for k-means clustering. It has parameter `centers =` that is used to define the number of groups.

```
# Plot original data in the 'weight'-'qsec space'
plot(mtcars$qsec, # qsec
     mtcars$wt,   # weight
     pch = 16,
     col = 'black',
     xlab = 'qsec: 1/4 mile time (s)',
     ylab = 'Weight (1000lbs)',
     main = 'Weight vs. qsec',
     ylim = c(1.5, 5.5))
```





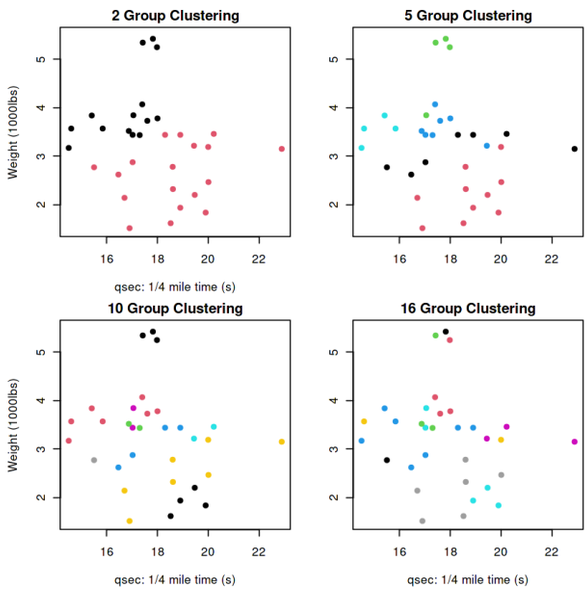

```r
# Track sum of squares error for each clustering model
errors = numeric(16)
centroids = seq(1, 16) # testing number of groups equal to 1,2,...,16

# Build models and record error
for (i in centroids) {
  km <- kmeans(mtcars, centers = i)
  errors[i] <- km$tot.withinss
}

# Display results
plot(centroids,
     errors,
     col = 'blue',
     type = 'l',
     xlab = 'Number of Centroids (Groups)',
     ylab = 'Sum Squared Error',
     main = 'Clustering mtcars')
```




```r
# Build models for visual comparison
kmc2 <- kmeans(mtcars, centers = 2)   # 2 groups
kmc5 <- kmeans(mtcars, centers = 5)   # 5 groups
kmc10 <- kmeans(mtcars, centers = 10) # 10 groups
kmc16 <- kmeans(mtcars, centers = 16) # 16 groups
```

```r
# Display the clustering results in the 'weight'-'qsec' space:
par(mfrow = c(2,2),            # plot locations 2x2
    mai = c(0.7,0.7,0.3,0.05)) # plot spacing (bottom, left, top, right)

# 2 group model
plot(mtcars$qsec,                      # qsec
     mtcars$wt,                        # weight
     pch = 16,
     col = kmc2$cluster,               # colour based on model cluster
     xlab = 'qsec: 1/4 mile time (s)',
     ylab = 'Weight (1000lbs)',
     main = '2 Group Clustering',
     ylim = c(1.5, 5.5))
```

```r
# 5 group model
plot(mtcars$qsec,
     mtcars$wt,
     pch = 16,
     col = kmc5$cluster,
     xlab = '',
     ylab = '',
     main = '5 Group Clustering',
     ylim = c(1.5, 5.5))

# 10 group model
plot(mtcars$qsec,
     mtcars$wt,
     pch = 16,
     col = kmc10$cluster,
     xlab = 'qsec: 1/4 mile time (s)',
     ylab = 'Weight (1000lbs)',
     main = '10 Group Clustering',
     ylim = c(1.5, 5.5))

# 16 group model
plot(mtcars$qsec,
     mtcars$wt,
     pch = 16,
     col = kmc16$cluster,
     xlab = 'qsec: 1/4 mile time (s)',
     ylab = '',
     main = '16 Group Clustering',
     ylim = c(1.5, 5.5))
```



The same four groups that were displayed for the Python clustering are displayed here in R. Even though both R and Python models do have randomness incorporated (choosing location of centroids) one can notice the the model results are very similar.

8.6 - Time Series

Time series modelling or forecasting is a supervised learning method which seeks to make predictions into the future. These models are made for historical data sets which have time-stamped data points. Using the R datasets package one can import the "Air passengers" data set. As per the datasets documentation page [20] The R Datasets Package, the Air passengers data set is "The classic Box & Jenkins airline data. Monthly totals of international airline passengers, 1949 to 1960."

8.6.1 - Python Time Series

The statsmodles modules is also used here for its time series tools; specifically, the `seasonal_decompose()` and `ARIMA()` functions.

```python
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.arima.model import ARIMA
```

The air passengers data can be imported as a CSV file with the pandas function `read_csv()`. We then convert the CSV file to a time series using the pandas function `Series()`.

```python
data = pd.read_csv('AirPassengers.csv') # import air passenger data
airpass = pd.Series(list(data['x']),
                    index = pd.date_range("1/1/1949",
                                          freq = "M",
                                          periods = 144))
airpass.head()
```

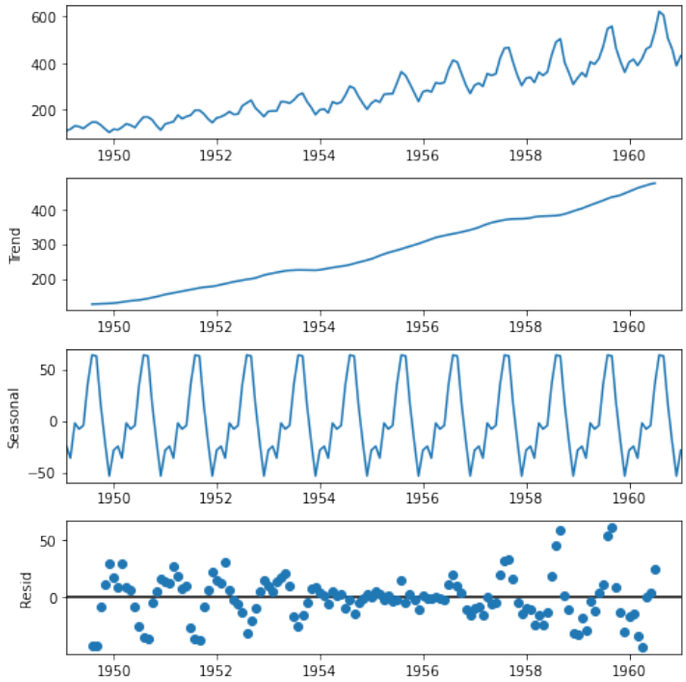
```
1949-01-31    112
1949-02-28    118
1949-03-31    132
1949-04-30    129
1949-05-31    121
Freq: M, dtype: int64
```

The `seasonal_decompose()` function will break the time series data into three separate parts in which their summation equals the true data. The decomposition is of the form:

$$data = trend + seasonal + residual$$

Looking at the plots these parts can be thought of as an average component plus a cyclical component plus the error.

```
decomp = seasonal_decompose(airpass, model = 'additive', period = 12)
decomp.plot()
plt.show()
```



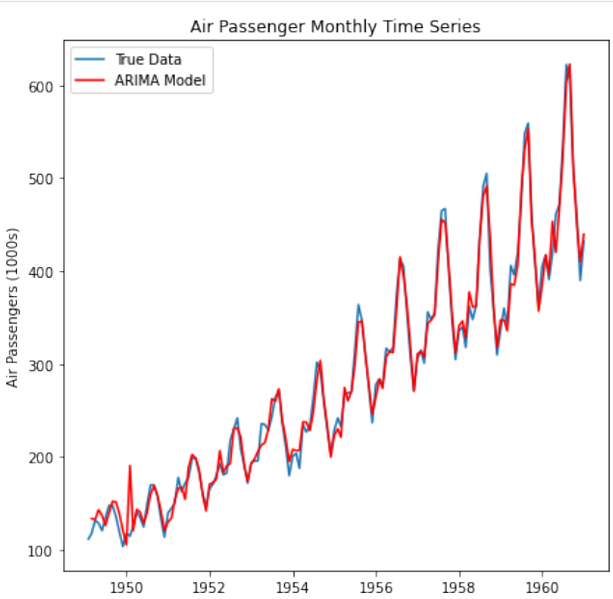
The `ARIMA()` function is used to fit an arima time series model. Note that the parameters provided for `order`, `seasonal_order` and `enforce_stationarity` can be changed to make the model behave differently.

```
ts = ARIMA(airpass,
           order = (2,1,1),
           seasonal_order = (0,1,0,12),
           enforce_stationarity = False)
ts_fit = ts.fit()
```

```
/home/slmcconv/.local/lib/python3.7/site-packages/statsmodels/base/model.py:606: ConvergenceWarning: Maximum Likelihood optimization fail
ed to converge. Check mle_retvals
  ConvergenceWarning)
```

Once a model is fit to the data one can plot its fitted values to compare to true values.

```
# Actual vs Fitted
fit = ts_fit.predict(1)

plt.plot(airpass, label = 'True Data')
plt.title('Air Passenger Monthly Time Series')
plt.xlabel('Time')
plt.ylabel('Air Passengers (1000s)')
plt.plot(fit, color = 'red', label = 'ARIMA Model')
plt.legend()
plt.show()
```



The time series model can then be used to make predictions into the future. This is done with the `get_forecast()` function. One can then use function `predicted_mean` and `conf_int` to get the predicted forecast along with the 95% confidence interval.

```
fore = ts_fit.get_forecast(12)
forecast_mean = fore.predicted_mean
forecast_low = fore.conf_int(alpha=0.05)['lower y']
forecast_high = fore.conf_int(alpha=0.05)['upper y']

plt.plot(airpass[120:144],
         label = 'True Data')
plt.plot(fit[119:143],
         color = 'red',
         label = 'ARIMA Model')
plt.plot(forecast_mean,
         color = 'red',
         linestyle = 'dashed',
         label = 'Predicted')
plt.fill_between(forecast_low.index,
                 forecast_low,
                 forecast_high,
                 alpha = 0.3,
                 color = 'red',
                 label = '95% CI')
plt.title('Air Passenger Monthly Time Series')
plt.xlabel('Time')
plt.ylabel('Air Passengers (1000s)')
plt.xticks(['1959', '1960', '1961', '1962'])
plt.legend()
plt.show()
```

8.6.2 - R Forecast package

To create ARIMA time series models in R one can use the forecast package. As per the forecast documentation page [23] Automatic Time Series Forecasting: The forecast Package for R "The R package forecast provides methods and tools for displaying and analysing univariate time series forecasts including exponential smoothing via state space models and automatic ARIMA modelling." We use the package here for its useful functions for decomposition, `stl()`, and ARIMA time series modelling, `auto.arima()`. The R forecast package also has an alternative reference to its formal white paper which can be found at [24] forecast: Forecasting Functions for Time Series and Linear Models for those who are interested in a more detailed view.

The forecast package can be imported with the call `library(forecast)` as with all pervious R packages. It should be noted that version 8.12 of the forecast package is being used.

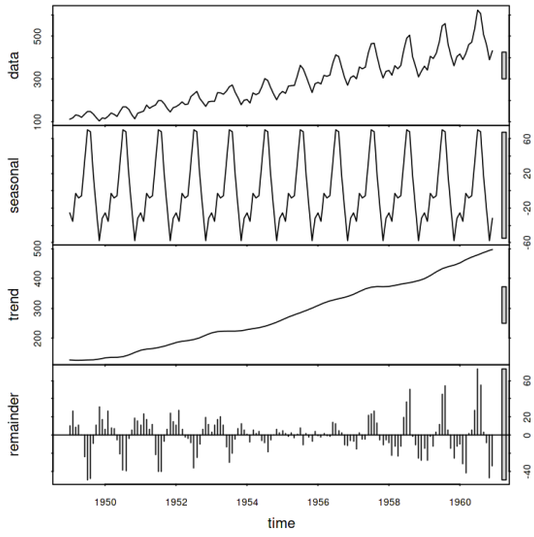
```
library(forecast)
packageVersion("forecast")
```

```
[1] '8.12'
```

8.6.3 R Time Series

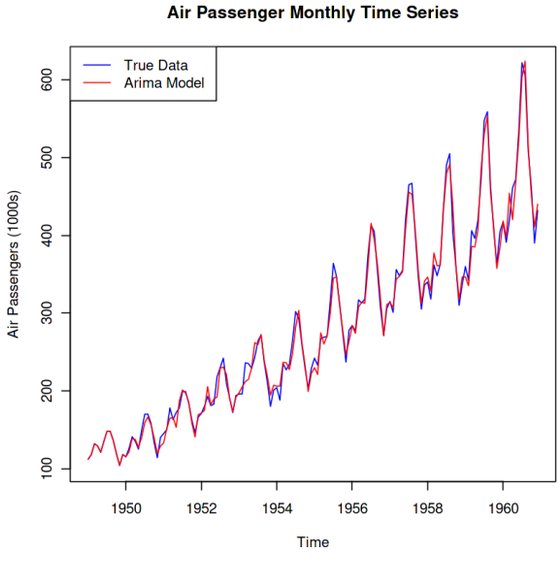
Using the `stl()` function from forecast we can create a decomposition plot similar to what was done in Python.

```
# decmopose time series
decomp <- stl(AirPassengers, s.window = 'period')
plot(decomp)
```



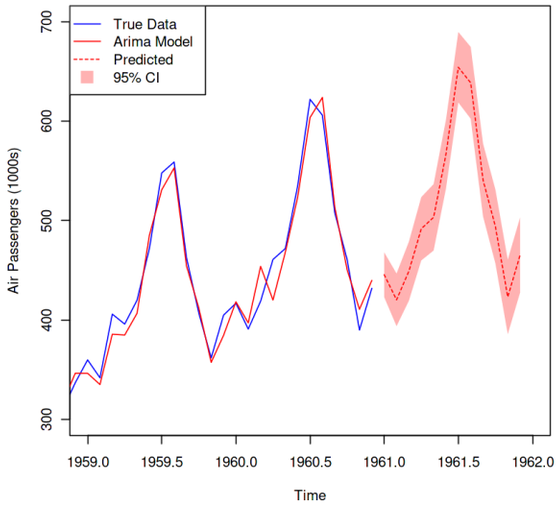
Using the function `auto.arima()` one can fit the ARIMA time series model.

```
# plot data
plot(AirPassengers,
     ylab = 'Air Passengers (1000s)',
     main = 'Air Passenger Monthly Time Series',
     col = 'blue')

fit <- auto.arima(AirPassengers)
points(fit$fitted, col = 'red', type = 'l')
legend('topleft',
       legend = c('True Data', 'Arima Model'),
       col = c('blue', 'red'),
       lwd = 1)
```



Similar to Python, the function `forecast()` is used to make predictions of the time series into the future. It also includes values for a 95% confidence interval which is shown in the plot below.

```
forecast <- forecast(fit, 12)

plot(AirPassengers,
     ylab = 'Air Passengers (1000s)',
     main = 'Air Passenger Monthly Time Series',
     col = 'blue',
     ylim = c(300, 700),
     xlim = c(1959, 1962))
points(fit$fitted, col = 'red', type = 'l')

points(forecast$mean, type ='l', col = 'red', lty = 2)
x <- seq(1961, 1962 - 1/12, length.out = 12)
polygon(c(x,rev(x)),
        c(forecast$upper[,2], rev(forecast$lower[,2])),
        col = adjustcolor("red", 0.3),
        border = NA)
legend('topleft',
       legend = c('True Data', 'Arima Model', 'Predicted', '95% CI'),
       col = c('blue', 'red', 'red', adjustcolor('red', 0.3)),
       lwd = c(1, 1, 1, NA),
       lty = c(1, 1, 2, NA),
       pch = c(NA, NA, NA, 15),
       pt.cex = c(1, 1, 1, 2))
```



8.7 - Neural Networks in Python

Since Python is a general purpose coding language, it provides much more flexibility or control over what you can accomplish with given modules compared to R and its packages. For this reason, it is generally accepted that Python is better suited than R for deep learning applications such as neural networks. This section provides a neural network template class built on top of the Python module PyTorch. The template can be changed and tweaked to create a wide array of neural network architectures which one might find extremely useful.

Neural networks are a very powerful and flexible type of machine learning model. Typically these models are best suited for extremely large data sets and best run with high value computational resources. Fortunately with the advancement in modern computers these types of models are now accessible and can be built on virtually any modern laptop processor.

There are many types of neural network models such as convolutional neural networks or recurrent neural networks. This section chooses to focus on neural networks in their simplest form which is a fully connected neural network. This section shows how the neural network model can be built to solve a classification problem.

8.7.1 - Python PyTorch Module

When building neural networks in Python the PyTorch module is used exclusively. PyTorch provides many tools that make the neural network design process much easier. As per the PyTorch documentation page [21] PyTorch: An Imperative Style High-Performance Deep Learning Library "PyTorch provides the elegantly designed modules and classes `torch.nn` , `torch.optim` , `Dataset` , and `DataLoader` to help you create and train neural networks". This section provides a small tutorial on PyTorch covering tensors, automatic differentiation, data loaders and network classes. A much more detailed tutorial on the specific tools can be found on [21].

PyTorch can be imported with the call `import torch` once downloaded. It should be noted that version 1.11.0+cu102 of PyTorch is being used.

```
import torch
print(torch.__version__)
```

```
1.11.0+cu102
```

8.7.1.1 - PyTorch Tensors

The PyTorch package deals with tensor objects. These are specialized data structures which behave very similarly to NumPy data structures. One can think of vectors as 1-dimensional tensors, ie. tensors that have size (1, n) or (n, 1) for some positive integer n. Matrices can be thought of the same way, as tensors with size (n, m) for some positive integers n and m.

One should notice that almost any NumPy array function will also have a Torch tensor counterpart which behaves exactly the same way.

8.7.1.2 - Vector Tensors

```
# arange(n) gives an tensor of size n with elements from 0 to n-1
X = torch.arange(3)
X
```

```
tensor([0, 1, 2])
```

```
# ones(n) gives tensor of size n with all elements equal to 1
X = torch.ones((1,3)) # 1 row 3 columns
X
```

```
tensor([[1., 1., 1.]])
```

```
# zeros(n) gives tensor of size n with all elements equal to 0
X = torch.zeros((3,1)) # 3 rows 1 column
X
```

```
tensor([[0.],
        [0.],
        [0.]])
```

```
# tensor() allows for manual tensor creation
X = torch.tensor([0,0,0.]) # "0." enforces floating point
X
```

```
tensor([0., 0., 0.])
```

8.7.1.3 - Indexing Vector Tensors

```
X = torch.tensor([1,2,3,4])
X
```

```
tensor([1, 2, 3, 4])
```

```
X[0] # first element
```

```
tensor(1)
```

```
X[3] # last element
```

```
tensor(4)
```

```
X[0:3] # use colon sytax n:m to get elements from postion n to m-1
```

```
tensor([1, 2, 3])
```

```
X[[0,1,2,3,0,0,1,1]] # insert array of indices to get any elements
```

```
tensor([1, 2, 3, 4, 1, 1, 2, 2])
```

8.7.1.4 - Matrix Tensors

```
M = torch.eye(2) # idenity matrix as tensor
M
```

```
tensor([[1., 0.],
        [0., 1.]])
```

```
M = torch.tensor([[1,2],[3,4]]) # manual matrix creation
M
```

```
tensor([[1, 2],
        [3, 4]])
```

```
m = torch.tensor([1,2,3,4]) # matrix tensor from vector tensor
M = m.reshape(2,2)
M
```

```
tensor([[1, 2],
        [3, 4]])
```

8.7.1.5 - Indexing Matrix Tensors

```
M[0,0] # first row, first column (top left)
```

```
tensor(1)
```

```
M[1,1] # last row, last colummn (bottom right)
```

```
tensor(4)
```

```
M[:,0] # first column
```

```
tensor([1, 3])
```

```
M[1,:] # last row
```

```
tensor([3, 4])
```

```
M[[0,1,1], [1,1,1]] # specific elements
```

```
tensor([2, 4, 4])
```

```
M[[[0],[1],[1]], [1,1,1]] # specific rows and elements
```

```
tensor([[2, 2, 2],
        [4, 4, 4],
        [4, 4, 4]])
```

8.7.1.6 - Vector Tensor Operations

```
X1 = torch.tensor([1,2,3])
X2 = torch.tensor([2,2,2])
X1, X2
```

```
(tensor([1, 2, 3]), tensor([2, 2, 2]))
```

```
X3 = X1 * X2 # element-wise multiplication
X3
```

```
tensor([2, 4, 6])
```

```
X4 = X1 @ X2 # vector product
X4
```

```
tensor(12)
```

```
X5 = X1 + X2 # vector addition
X5
```

```
tensor([3, 4, 5])
```

```
X6 = X1 + 2 # vector/scalar addition
X6
```

```
tensor([3, 4, 5])
```

```
X7 = X1 * 2 # vector/scalar multiplication
X7
```

```
tensor([2, 4, 6])
```

8.7.1.7 - Matrix Tensor Operations

```
X = torch.tensor([[1,3,5],[2,4,6]])
X
```

```
tensor([[1, 3, 5],
        [2, 4, 6]])
```

```
M1 = X * X # element-wise multiplication
M1
```

```
tensor([[ 1,  9, 25],
        [ 4, 16, 36]])
```

```
M2 = X.T @ X # matrix product
M2
```

```
tensor([[ 5, 11, 17],
        [11, 25, 39],
        [17, 39, 61]])
```

```
M3 = X + X # matrix/matrix addition
M3
```

```
tensor([[ 2,  6, 10],
        [ 4,  8, 12]])
```

```
M4 = X + 2 # matrix/scalar addition
M4
```

```
tensor([[3, 5, 7],
        [4, 6, 8]])
```

```
M5 = X*2 # matrix/scalar multiplication
M5
```

```
tensor([[ 2,  6, 10],
        [ 4,  8, 12]])
```

8.7.1.8 - Matrix-Vector Tensor Operations

```
x = torch.tensor([1,2])
x
```

```
tensor([1, 2])
```

```
M = X.T @ x # matrix/vector product
M
```

```
tensor([ 5, 11, 17])
```

```
M1 = X.T * x # multiplied across rows
M1
```

```
tensor([[ 1,  4],
        [ 3,  8],
        [ 5, 12]])
```

```
x2 = torch.tensor([1,2,3])
M2 = X * x2 # multiplied across columns
M2
```

```
tensor([[ 1,  6, 15],
        [ 2,  8, 18]])
```

```
M3 = X.T + x # addition is the same across rows of X
M3
```

```
tensor([[2, 4],
        [4, 6],
        [6, 8]])
```

```
M4 = X + x2 # addition is the same across columns of X
M4
```

```
tensor([[2, 5, 8],
        [3, 6, 9]])
```

8.7.1.9 - Higher Dimensional Tensors

Just as with NumPy, one can create multi-dimensional tensors of any arbitrary shape.

```
T = torch.ones(size = (2,2,2,2)) # Any dimension
T
```

```
tensor([[[[1., 1.],
          [1., 1.]],

         [[1., 1.],
          [1., 1.]]],


        [[[1., 1.],
          [1., 1.]],

         [[1., 1.],
          [1., 1.]]]])
```

8.7.1.10 - PyTorch Automatic Differentiation

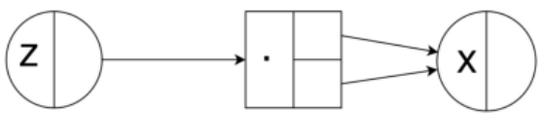
PyTorch has a built-in differentiation engine which allows one to compute gradients easily for any computational graph. Each tensor that one defines can be given the parameter `requires_grad = True` which creates a place in memory for that tensor to track is own gradient.

Consider the simple example:

$$
\begin{aligned}
x &= [x_1\ x_2] \in \mathbb{R}^2 \\
Z &= x \cdot x = x^T x = x_1^2 + x_2^2 \in \mathbb{R}
\end{aligned}
$$

$$
\Rightarrow \nabla_x Z = [\frac{\partial Z}{\partial x_1}\ \frac{\partial Z}{\partial x_2}] = [2x_1\ 2x_2] \in \mathbb{R}^2
$$

$$
\therefore\ x = [1\ 1]\ \Rightarrow\ Z = 2\ and\ \nabla_x Z = [2\ 2]
$$




```python
x = torch.ones(2, requires_grad = True) # define x
z = x @ x                               # define z
x, z
```

```
(tensor([1., 1.], requires_grad=True), tensor(2., grad_fn=<DotBackward0>))
```

Calling `backward()` and then displaying the gradients with `x.grad()` one can confirm that the auto differentiation calculations are correct.

```python
z.backward() # backprop to get graidents (only call this once after evaluating z)
x.grad       # gradient of z wrt x
```

```
tensor([2., 2.])
```

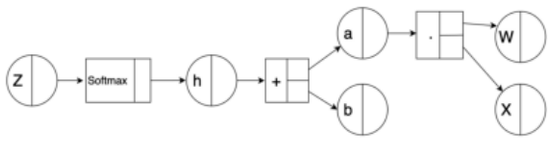
Consider another example:

$$
\begin{aligned}
x &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2 \\
W &= \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \in \mathbb{R}^{2\times 2} \\
b &= \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \in \mathbb{R}^2 \\
h &= Wx + b = \begin{bmatrix} W_{11}x_1 + W_{12}x_2 + b_1 \\ W_{21}x_1 + W_{22}x_2 + b_2 \end{bmatrix} = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} \in \mathbb{R}^2 \\
Z &= Softmax(h) = \begin{bmatrix} \frac{e^{h_1}}{e^{h_1}+e^{h_2}} \\ \frac{e^{h_2}}{e^{h_1}+e^{h_2}} \end{bmatrix} = \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} \in \mathbb{R}^2
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow \nabla_b Z_1 &= \begin{bmatrix} \frac{\partial Z_1}{\partial b_1} \\ \frac{\partial Z_1}{\partial b_2} \end{bmatrix} = \begin{bmatrix} \frac{e^{h_1+h_2}}{(e^{h_1}+e^{h_2})^2} \\ \frac{-e^{h_1+h_2}}{(e^{h_1}+e^{h_2})^2} \end{bmatrix} \in \mathbb{R}^2 \\
\nabla_b Z_2 &= \begin{bmatrix} \frac{\partial Z_2}{\partial b_1} \\ \frac{\partial Z_2}{\partial b_2} \end{bmatrix} = \begin{bmatrix} \frac{-e^{h_1+h_2}}{(e^{h_1}+e^{h_2})^2} \\ \frac{e^{h_1+h_2}}{(e^{h_1}+e^{h_2})^2} \end{bmatrix} \in \mathbb{R}^2 \\
\nabla_W Z_1 &= \begin{bmatrix} \frac{\partial Z_1}{\partial W_{11}} & \frac{\partial Z_1}{\partial W_{21}} \\ \frac{\partial Z_1}{\partial W_{21}} & \frac{\partial Z_1}{\partial W_{22}} \end{bmatrix} = \begin{bmatrix} \frac{x_1 e^{h_1+h_2}}{(e^{h_1}+e^{h_2})^2} & \frac{x_2 e^{h_1+h_2}}{(e^{h_1}+e^{h_2})^2} \\ \frac{-x_1 e^{h_1+h_2}}{(e^{h_1}+e^{h_2})^2} & \frac{-x_2 e^{h_1+h_2}}{(e^{h_1}+e^{h_2})^2} \end{bmatrix} \in \mathbb{R}^{2\times 2} \\
\nabla_W Z_2 &= \begin{bmatrix} \frac{\partial Z_2}{\partial W_{11}} & \frac{\partial Z_2}{\partial W_{12}} \\ \frac{\partial Z_2}{\partial W_{21}} & \frac{\partial Z_2}{\partial W_{22}} \end{bmatrix} = \begin{bmatrix} \frac{-x_1 e^{h_1+h_2}}{(e^{h_1}+e^{h_2})^2} & \frac{-x_2 e^{h_1+h_2}}{(e^{h_1}+e^{h_2})^2} \\ \frac{x_1 e^{h_1+h_2}}{(e^{h_1}+e^{h_2})^2} & \frac{x_2 e^{h_1 2+h_2}}{(e^{h_1}+e^{h_2})^2} \end{bmatrix} \in \mathbb{R}^{2\times 2}
\end{aligned}
$$

$$\therefore x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \; W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \; b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\Rightarrow h = \begin{bmatrix} 3 \\ 3 \end{bmatrix}, \; Z = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}, \; \nabla_b Z_1 = \begin{bmatrix} \frac{1}{4} \\ \frac{-1}{4} \end{bmatrix}, \nabla_b Z_2 = \begin{bmatrix} \frac{-1}{4} \\ \frac{1}{4} \end{bmatrix}, \; \nabla_W Z_1 = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} \\ \frac{-1}{4} & \frac{-1}{4} \end{bmatrix} \; and \; \nabla_W Z_2 = \begin{bmatrix} \frac{-1}{4} & \frac{-1}{4} \\ \frac{1}{4} & \frac{1}{4} \end{bmatrix}$$



It should be noted that $a = Wx$ is just a place holder in the computational graph.

```
x = torch.ones(2, requires_grad = False)            # define x
b = torch.ones(2, requires_grad = True)             # define b
W = torch.ones(size = (2,2), requires_grad = True)  # define W
x, b,  W
```

```
(tensor([1., 1.]),
 tensor([1., 1.], requires_grad=True),
 tensor([[1., 1.],
         [1., 1.]], requires_grad=True))
```

```
h = W @ x + b                        # define h
softmax = torch.nn.Softmax(dim = 0) # softmax function
z = softmax(h)                       # define z
h, z
```

```
(tensor([3., 3.], grad_fn=<AddBackward0>),
 tensor([0.5000, 0.5000], grad_fn=<SoftmaxBackward0>))
```

```
# Backprop to get gradients wrt to z1 (can only call wrt to a scalar value)
z.backward(torch.FloatTensor([1, 0]), retain_graph = True)
```

Again one can use `backward()` and then access the gradients to confirm the differentiation results are correct.

```
W.grad.data # gradient of z1 wrt W
```

```
tensor([[ 0.2500,  0.2500],
        [-0.2500, -0.2500]])
```

```
b.grad.data # gradient of z1 wrt b
```

```
tensor([ 0.2500, -0.2500])
```

The gradients wrt Z_1 are cleared and set to 0 so the new computation of gradient wrt Z_2 can be computed and updated.

```
# Clear previous gradients
W.grad.data.zero_()
b.grad.data.zero_()

# Backprop to get graidents wrt to z2
z.backward(torch.FloatTensor([0, 1]), retain_graph = True)
```

Again one can confirm that the auto differentiation is correct.

```
W.grad # gradient of z2 wrt W
```

```
tensor([[-0.2500, -0.2500],
        [ 0.2500,  0.2500]])
```

```
b.grad # gradient of z2 wrt b
```

```
tensor([-0.2500,  0.2500])
```

8.7.1.11 - PyTorch Datasets and Data Loaders

The torchvision module is imported in order to download the MNIST data set. This is done using the function `torchvision.datasets.MNIST()`. It should be noted that version 0.12.0+cu102 is being used.

```
import torchvision
print(torchvision.__version__)
```

```
0.12.0+cu102
```

The MNIST data consists of handwritten digits between 0 and 9. These images are in greyscale form with pixel values between 0 and 1. Each image here is imported with dimensions (28 x 28) and then flattened to a 282=784 sized vector of dimension (1 x 784). The flattened data is what gets passed into the neural network model.

$$IMAGE = \begin{bmatrix} x_{1,1} & \cdots & x_{1,28} \\ \vdots & \ddots & \vdots \\ x_{28,1} & \cdots & x_{28,28} \end{bmatrix}, \; x_{i,j} \in [0,1] \; \forall \, i,j = 1,2,\ldots,28 \; \in \mathbb{R}^{28 \times 28}$$

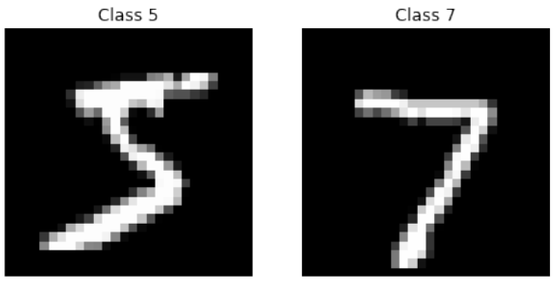
$$IMAGE_{flat} = [x_{1,1} \quad \cdots \quad x_{1,28} \quad x_{2,1} \quad \cdots \quad x_{2,28} \quad \cdots \quad \cdots \quad x_{28,1} \quad \cdots \quad x_{28,28}] \in \mathbb{R}^{784}$$

This flattening technique is useful for non-convolution networks as one can use basic linear connection layers. It should be noted that flattening the image causes an information loss in the data. The information of pixels above, below and diagonal are 'lost' when flattening and one is left with only the information of pixels to the left and right. Note: if the data are kept in matrix form of size (28 x 28) then one will need convolution layers in the neural network model.

```
# Set image size, can be 14x14 if desired
img_size = 28

# Get MNIST training data
MNIST_train = torchvision.datasets.MNIST(
    './files/',
    train = True,                                                   # training data
    download = True,
    transform = torchvision.transforms.Compose([
        torchvision.transforms.Resize((img_size, img_size)),        # get (img_size X img_size)
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Lambda((lambda x: torch.flatten(x))) # flatten to vector (img_size^2 X 1)
    ])
)
# Set training subest
train_size = 1500
ds_train = torch.utils.data.Subset(MNIST_train, range(train_size))

# Get MNIST test data
MNIST_test = torchvision.datasets.MNIST(
    './files/',
    train = False,                                                  # test data
    download = True,
    transform = torchvision.transforms.Compose([
        torchvision.transforms.Resize((img_size, img_size)),        # get (img_size X img_size)
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Lambda((lambda x: torch.flatten(x))) # flatten to vector (img_size^2 X 1)
    ])
)
# Set testing subset
test_size = 250
ds_test = torch.utils.data.Subset(MNIST_test, range(test_size))
```

Here we get the first sample in both the training and testing subsets of the MNIST data.

```
# First training data entry
x_train_1 = MNIST_train[0][0] # data
t_train_1 = MNIST_train[0][1] # target

# Display first training data observation
plt.subplot(1,2,1)
plt.imshow(x_train_1.reshape((img_size, img_size)).detach().numpy(),
           cmap = 'gray')
plt.axis('off')
plt.title(f'Class {t_train_1}')

# First test data entry
x_test_1 = MNIST_test[0][0] # data
t_test_1 = MNIST_test[0][1] # target

# Display first test data observation
plt.subplot(1,2,2)
plt.imshow(x_test_1.reshape((img_size, img_size)).detach().numpy(),
           cmap = 'gray')
plt.axis('off')
plt.title(f'Class {t_test_1}')
plt.show()
```




```
# Number of data points in each full train/test set
print('Full Number of Training Observations:', len(MNIST_train)) # train
print('Full Number of Test Observations:', len(MNIST_test))      # test

# Number of data points in each smaller train/test set
print('Reduced Number of Training Observations:', len(ds_train)) # train
print('Reduced Number of Test Observations:', len(ds_test))      # test
```

```
Full Number of Training Observations: 60000
Full Number of Test Observations: 10000
Reduced Number of Training Observations: 1500
Reduced Number of Test Observations: 250
```

Here the PyTorch function `torch.utils.data.DataLoader()` is used to create batches for the MNIST data. The data loaders created here have batch sizes of 50 and implemented a random shuffle for each batch gathered.

```
# Batched for training
train_batch_size = 50
train_dl = torch.utils.data.DataLoader(ds_train,                      # MNIST training data
                                       batch_size = train_batch_size, # batches of size train_batch_size
                                       shuffle = True)                # randomly shuffle to create batches
# No batch, full test set
test_full = torch.utils.data.DataLoader(ds_test,                    # MNIST test data
                                        batch_size = len(ds_test), # 1 batch with all test data
                                        shuffle = False)           # no random shuffle needed with 1 batch
```

Looking at the shape of a given batch for both input and target one can confirm that we have 50 samples of correct input size (784) and correct output size (1).

```
# Get one batch of training data
batch_data, batch_targets = next(iter(train_dl))

# Check dimesnions of batches (batch size X data size)
print('Training batch data size:', batch_data.size())
print('Training batch target size:', batch_targets.size())
```

```
Training batch data size: torch.Size([50, 784])
Training batch target size: torch.Size([50])
```

8.7.2 - Building a Neural Network with PyTorch

There are many possible ways to define and train a neural network model. It is shown here how to create a neural network Python class that includes the model itself along with a learning process all defined within the class. The learning process inside this class will need to function with the data loaders previously defined. One must also supply a loss function and a PyTorch optimization method. The class defined below is not set up to run on a GPU but can easily be changed to do so. This can be done by defining a GPU device and then sending data and targets to that device for each batch in the training/evaluation loops.

The network created below is built with five layers each with linear connections. This is under the section of `self.network` where the PyTorch functions `torch.nn.Sequential`, `torch.nn.Linear`, `torch.nn.ReLU()` and `torch.nn.Softmax` are all used.

```
class NeuralNet(torch.nn.Module):

    # nn_model = NeuralNet(img_size = 28)
    #
    # Create a fully connected neural network model for imput images of size (img_size X img_size).
    #
    # Input:
    #     img_size -> size of the input images, input smaples will be (img_size^2) vectors (tensors).
    #
    # Usage:
    #     nn_model = NeuralNet() -> defines model
    #     y = nn_model(x)        -> returns ouput

    # Define model
    def __init__(self, img_size = 28):

        # Model attributes
        self.img_size = img_size # image size
        self.train_losses = []   # track training loss
        self.test_losses = []    # track test loss
        self.train_acc = []      # track training classification accuracy
        self.test_acc = []       # track test classification accuracy
        super().__init__()

        # Build model architecture
        self.network = torch.nn.Sequential(
            torch.nn.Linear(self.img_size ** 2, 392), # layer 1 (784 -> 392)
            torch.nn.ReLU(),                          # ouput 1
            torch.nn.Linear(392, 196),                # layer 2 (392 -> 196)
            torch.nn.ReLU(),                          # output 2
            torch.nn.Linear(196, 98),                 # layer 3 (196 -> 98)
            torch.nn.ReLU(),                          # output 3
            torch.nn.Linear(98, 49),                  # layer 4 (98 -> 49)
            torch.nn.ReLU(),                          # output 4
            torch.nn.Linear(49, 10),                  # final layer (49 -> 10)
            torch.nn.Softmax(dim = 1)                 # final output
        )

    # Define forward pass of the model
    def forward(self, x):

        return self.network(x)

    # Define learning process
    def learn(self, dl_train, test=False, dl_test=None, epochs=20, optim=None, loss_f=None):

        # nn_model.learn(dl_train, test=False, dl_test=None, epochs=20, optim=None, loss_f=None)
        #
        # Trains the neural network model on the dataset given by the dataloader dl_train. Evaluates
        # the model on both the training data and test data (if test = True is specified).
        #
        # Inputs:
        #      dl_train -> training data loader
        #      test     -> logical value indicating if test data is to be evaluated while training
        #      dl_test  -> test data loader
        #      epochs   -> number of epochs to train
        #      optim    -> PyTorch optimizer function
        #      loss_f   -> PyTorch loss function
```

```
    # Check for optimizer and loss function
    if optim is None or loss_f is None:
        print('Specify optimizer and loss function.')
        return

    # Training which includes evaluation on test data
    if test:
        for epoch in range(epochs):

            train_loss = 0. # track training batch loss
            train_accu = 0. # track training batch accuracy

            # For each batch in the training data
            for x, t in dl_train:
                y = self(x)         # forward pass
                loss = loss_f(y, t) # get error (loss)
                optim.zero_grad()   # zero the gradients
                loss.backward()     # push current error down gradients
                optim.step()        # optimization step in gradient direction

                # update batch loss
                train_loss += loss.item() * len(t)

                # update batch accuracy
                with torch.no_grad():
                    train_accu += torch.sum((t == torch.argmax(y, dim = 1)) * 1) / len(t)

            # Track average training loss and accurcay for entire epoch
            self.train_losses.append(train_loss/len(dl_train.dataset))
            self.train_acc.append(train_accu/len(dl_train))

            test_loss = 0. # track test batch loss
            test_accu = 0. # track test batch accuracy

            # For each batch in the test data (no need for gradient info)
            with torch.no_grad():
                for x, t in dl_test:
                    y2 = self(x)          # forward pass
                    loss2 = loss_f(y2, t) # get error (loss)

                    # update batch loss
                    test_loss += loss2.item() * len(t)

                    # update batch accuracy
                    test_accu += torch.sum((t == torch.argmax(y2, dim = 1)) * 1) / len(t)

                # Track average test loss and accuracy for entire epoch
                self.test_losses.append(test_loss/len(dl_test.dataset))
                self.test_acc.append(test_accu/len(dl_test))

        # Display average training and test loss by epoch
        plt.subplot(2, 1, 1)
        plt.plot(self.train_losses, label = 'Train')
        plt.plot(self.test_losses, label = 'Test')
        plt.xlabel('')
        plt.title('Network Training')
        plt.ylabel('Error')
        plt.legend()

        # Display average training and test accuracy by epoch
        plt.subplot(2, 1, 2)
        plt.plot(self.train_acc, label = 'Train')
        plt.plot(self.test_acc, label = 'Test')
        plt.xlabel('Epochs')
        plt.title('')
        plt.ylabel('Classification Accuracy')
        plt.legend()
        plt.show()

    # Training with no test data evaluation
    else:
        for epoch in range(epochs):

            tot_loss = 0. # track training batch loss
            acc = 0.      # track training batch accuracy

            # For each batch in the training data
            for x, t in dl_train:
                y = self(x)         # forward pass
                loss = loss_f(y, t) # get error (loss)
                optim.zero_grad()   # zero the gradients
                loss.backward()     # push current error down gradients
                optim.step()        # optimization step in gradient direction

                # update batch loss
                tot_loss += loss.item() * len(t)

                # update batch accuracy
                with torch.no_grad():
                    acc += torch.sum((t == torch.argmax(y, dim = 1)) * 1) / len(t)

            # Track average training loss and accurcay for entire epoch
            self.train_losses.append(tot_loss/len(dl_train.dataset))
            self.train_acc.append(acc/len(dl_train))
```

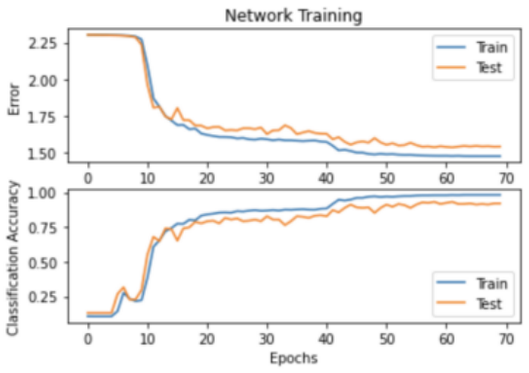
```
        # Display average training loss by epoch
        plt.subplot(2, 1, 1)
        plt.plot(self.train_losses)
        plt.xlabel('')
        plt.title('Network Training')
        plt.ylabel('Error')

        # Display average training accuracy by epoch
        plt.subplot(2, 1, 2)
        plt.plot(self.train_acc)
        plt.xlabel('Epochs')
        plt.title('')
        plt.ylabel('Classification Accuracy')
        plt.show()
```

8.7.3 - Training a Neural Network with PyTorch

Training the neural network model does not require much additional code as the learning function was defined within the neural network class above. One simple defines the network and then uses the function call `learn()` with given parameters to train. It should be noted that the PyTorch optimization method `torch.optim.SGD()` and the PyTorch loss function `torch.nn.CrossEntropyLoss()` are being used.

```
# Initialize model
nn_model = NeuralNet()
```

```
# Track training time (start)
start = time.time()

# Train model
nn_model.learn(dl_train = train_dl,                          # training data loader
               test = True,                                  # evaluation on test data
               dl_test = test_full,                          # test data loader
               epochs = 70,                                  # number of epochs
               optim = torch.optim.SGD(nn_model.parameters(), # optimize model parameters with SGD method
                                       lr = 0.15,            # learning rate for SGD
                                       momentum = 0.75),     # momentum rate for SGD
               loss_f = torch.nn.CrossEntropyLoss())         # loss function for learning

# Track training time (end)
print('No GPU Training Time:', round(time.time() - start, 2), '(s)')
```




```
No GPU Training Time: 827.46 (s)
```

8.7.3.1 - Training Neural Networks on a GPU

PyTorch was designed to be run on GPUs so its setup is very straightforward. Using the PyTorch function `device()` one can set the device type to `cuda` if a GPU is accessible. This can be checked with the PyTorch function `cuda.is_available()`.

```
# Set GPU device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

The exact same network class can be used here for running on a GPU. The only change one has to make is the additional code needed to send data to our given GPU. This is done by the addition of the call `x.to(device)` and `t.to(device)` for each {x, t} ({input, target}) in the data loaders previously designed. Within the learning function in the neural network class there is a loop `for x, t in dl_train` which iterates through data loader samples. The additional `to(device)` calls are made under this.

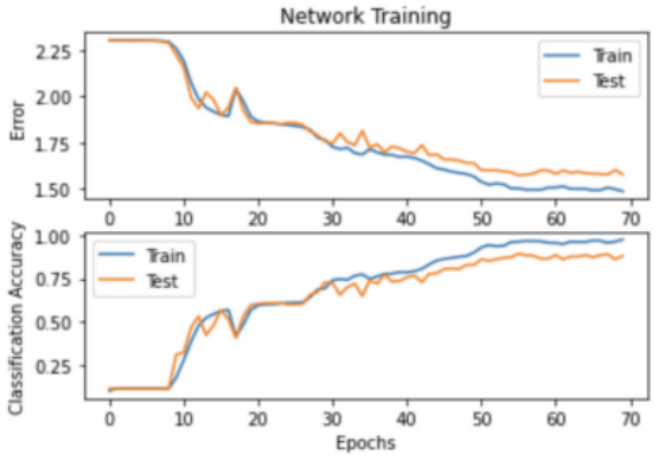
```
class NeuralNet(torch.nn.Module):

    # nn_model = NeuralNet(img_size = 28)
    #
    # Create a fully connected neural network model for imput images of size (img_size X img_size).
    #
    # Input:
    #     img_size -> size of the input images, input smaples will be (img_size^2) vectors (tensors).
    #
    # Usage:
    #     nn_model = NeuralNet() -> defines model
    #     y = nn_model(x)        -> returns ouput

    # Define model
    def __init__(self, img_size = 28):

        # Model attributes
```

```python
        self.img_size = img_size # image size
        self.train_losses = []   # track training loss
        self.test_losses = []    # track test loss
        self.train_acc = []      # track training classification accuracy
        self.test_acc = []       # track test classification accuracy
        super().__init__()

        # Build model architecture
        self.network = torch.nn.Sequential(
            torch.nn.Linear(self.img_size ** 2, 392), # layer 1 (784 -> 392)
            torch.nn.ReLU(),                          # ouput 1
            torch.nn.Linear(392, 196),                # layer 2 (392 -> 196)
            torch.nn.ReLU(),                          # output 2
            torch.nn.Linear(196, 98),                 # layer 3 (196 -> 98)
            torch.nn.ReLU(),                          # output 3
            torch.nn.Linear(98, 49),                  # layer 4 (98 -> 49)
            torch.nn.ReLU(),                          # output 4
            torch.nn.Linear(49, 10),                  # final layer (49 -> 10)
            torch.nn.Softmax(dim = 1)                 # final output
        )

    # Define forward pass of the model
    def forward(self, x):

        return self.network(x)

    # Define learning process
    def learn(self, dl_train, test=False, dl_test=None, epochs=20, optim=None, loss_f=None):

        # nn_model.learn(dl_train, test=False, dl_test=None, epochs=20, optim=None, loss_f=None)
        #
        # Trains the neural network model on the dataset given by the dataloader dl_train. Evaluates
        # the model on both the training data and test data (if test = True is specified).
        #
        # Inputs:
        #      dl_train -> training data loader
        #      test     -> logical value indicating if test data is to be evaluated while training
        #      dl_test  -> test data loader
        #      epochs   -> number of epochs to train
        #      optim    -> PyTorch optimizer function
        #      loss_f   -> PyTorch loss function

        # Check for optimizer and loss function
        if optim is None or loss_f is None:
            print('Specify optimizer and loss function.')
            return

        # Training which includes evaluation on test data
        if test:
            for epoch in range(epochs):

                train_loss = 0. # track training batch loss
                train_accu = 0. # track training batch accuracy

                # For each batch in the training data
                for x, t in dl_train:
                    x.to(device)        # send data to GPU
                    t.to(device)        # send targets to GPU
                    y = self(x)         # forward pass
                    loss = loss_f(y, t) # get error (loss)
                    optim.zero_grad()   # zero the gradients
                    loss.backward()     # push current error down gradients
                    optim.step()        # optimization step in gradient direction

                    # update batch loss
                    train_loss += loss.item() * len(t)

                    # update batch accuracy
                    train_accu += torch.sum((t == torch.argmax(y, dim = 1)) * 1) / len(t)

                # Track average training loss and accurcay for entire epoch
                self.train_losses.append(train_loss/len(dl_train.dataset))
                self.train_acc.append(train_accu/len(dl_train))

                test_loss = 0. # track test batch loss
                test_accu = 0. # track test batch accuracy

                # For each batch in the test data (no need for gradient info)
                with torch.no_grad():
                    for x, t in dl_test:
                        x.to(device)          # send data to GPU
                        t.to(device)          # send targets to GPU
                        y2 = self(x)          # forward pass
                        loss2 = loss_f(y2, t) # get error (loss)

                        # update batch loss
                        test_loss += loss2.item() * len(t)

                        # update batch accuracy
                        test_accu += torch.sum((t == torch.argmax(y2, dim = 1)) * 1) / len(t)

                    # Track average test loss and accuracy for entire epoch
                    self.test_losses.append(test_loss/len(dl_test.dataset))
                    self.test_acc.append(test_accu/len(dl_test))
```

```
            # Display average training and test loss by epoch
            plt.subplot(2, 1, 1)
            plt.plot(self.train_losses, label = 'Train')
            plt.plot(self.test_losses, label = 'Test')
            plt.xlabel('')
            plt.title('Network Training')
            plt.ylabel('Error')
            plt.legend()

            # Display average training and test accuracy by epoch
            plt.subplot(2, 1, 2)
            plt.plot(self.train_acc, label = 'Train')
            plt.plot(self.test_acc, label = 'Test')
            plt.xlabel('Epochs')
            plt.title('')
            plt.ylabel('Classification Accuracy')
            plt.legend()
            plt.show()

        # Training with no test data evaluation
        else:
            for epoch in range(epochs):

                tot_loss = 0. # track training batch loss
                acc = 0.      # track training batch accuracy

                # For each batch in the training data
                for x, t in dl_train:
                    x.to(device)        # send data to GPU
                    t.to(device)        # send targets to GPU
                    y = self(x)         # forward pass
                    loss = loss_f(y, t) # get error (loss)
                    optim.zero_grad()   # zero the gradients
                    loss.backward()     # push current error down gradients
                    optim.step()        # optimization step in gradient direction

                    # update batch loss
                    tot_loss += loss.item() * len(t)

                    # update batch accuracy
                    acc += torch.sum((t == torch.argmax(y, dim = 1)) * 1) / len(t)

                # Track average training loss and accurcay for entire epoch
                self.train_losses.append(tot_loss/len(dl_train.dataset))
                self.train_acc.append(acc/len(dl_train))

            # Display average training loss by epoch
            plt.subplot(2, 1, 1)
            plt.plot(self.train_losses)
            plt.xlabel('')
            plt.title('Network Training')
            plt.ylabel('Error')

            # Display average training accuracy by epoch
            plt.subplot(2, 1, 2)
            plt.plot(self.train_acc)
            plt.xlabel('Epochs')
            plt.title('')
            plt.ylabel('Classification Accuracy')
            plt.show()
```

The model can be trained in the exact same way because all the necessary GPU changes were made in the network class above. This time one can expect a giant increase in training speed.

```
# Initialize model
nn_model = NeuralNet()

# Track training time (start)
start = time.time()

# Train model
nn_model.learn(dl_train = train_dl,                          # training data loader
               test = True,                                  # evaluation on test data
               dl_test = test_full,                          # test data loader
               epochs = 70,                                  # number of epochs
               optim = torch.optim.SGD(nn_model.parameters(), # optimize model parameters with SGD method
                                       lr = 0.15,            # learning rate for SGD
                                       momentum = 0.75),     # momentum rate for SGD
               loss_f = torch.nn.CrossEntropyLoss())         # loss function for learning

# Track training time (end)
print('GPU Training Time:', round(time.time() - start, 2), '(s)')
```




```
GPU Training Time: 23.09 (s)
```

Training the same exact neural network model on a GPU allows for much faster training completion. One can see that the training time decreased from 827 seconds to 23 seconds, over 35 times faster! Training on a GPU can allow one to increased the size of training and testing datasets while maintaining a lower time for training completion.

It should be noted that training run times will vary depending on the machine one uses. The main take-away is that training on a GPU with PyTorch is both 1) very easy to implement and 2) much faster.

---

[11] Seabold, Skipper, and Josef Perktold, 2010. statsmodels: Econometric and Statistical Modeling with Python, [link]
[19] McKinney, W. & others, 2010. pandas: Data Structures for Statistical Computing in Python, [link]
[20] R Core Team and contributors worldwide, 2022. The R Datasets Package, [link]
[21] Paszke, A. et al., 2019. PyTorch: An Imperative Style High-Performance Deep Learning Library, [link]
[22] Pedregosa et al., 2011. Scikit-learn: Machine Learning in Python, [link]
[23] Hyndman R, Athanasopoulos G, Bergmeir C, Caceres G, Chhay L, O'Hara-Wild M, Petropoulos F, Razbash S, Wang E, Yasmeen F, 2020. forecast: Forecasting Functions for Time Series and Linear Models, [link]
[24] Hyndman RJ, Khandakar Y, 2008. Automatic Time Series Forecasting: The forecast Package for R, [link]
[25] Terry Therneau and Beth Atkinson, 2019. rpart: Recursive Partitioning and Regression Trees, [link]