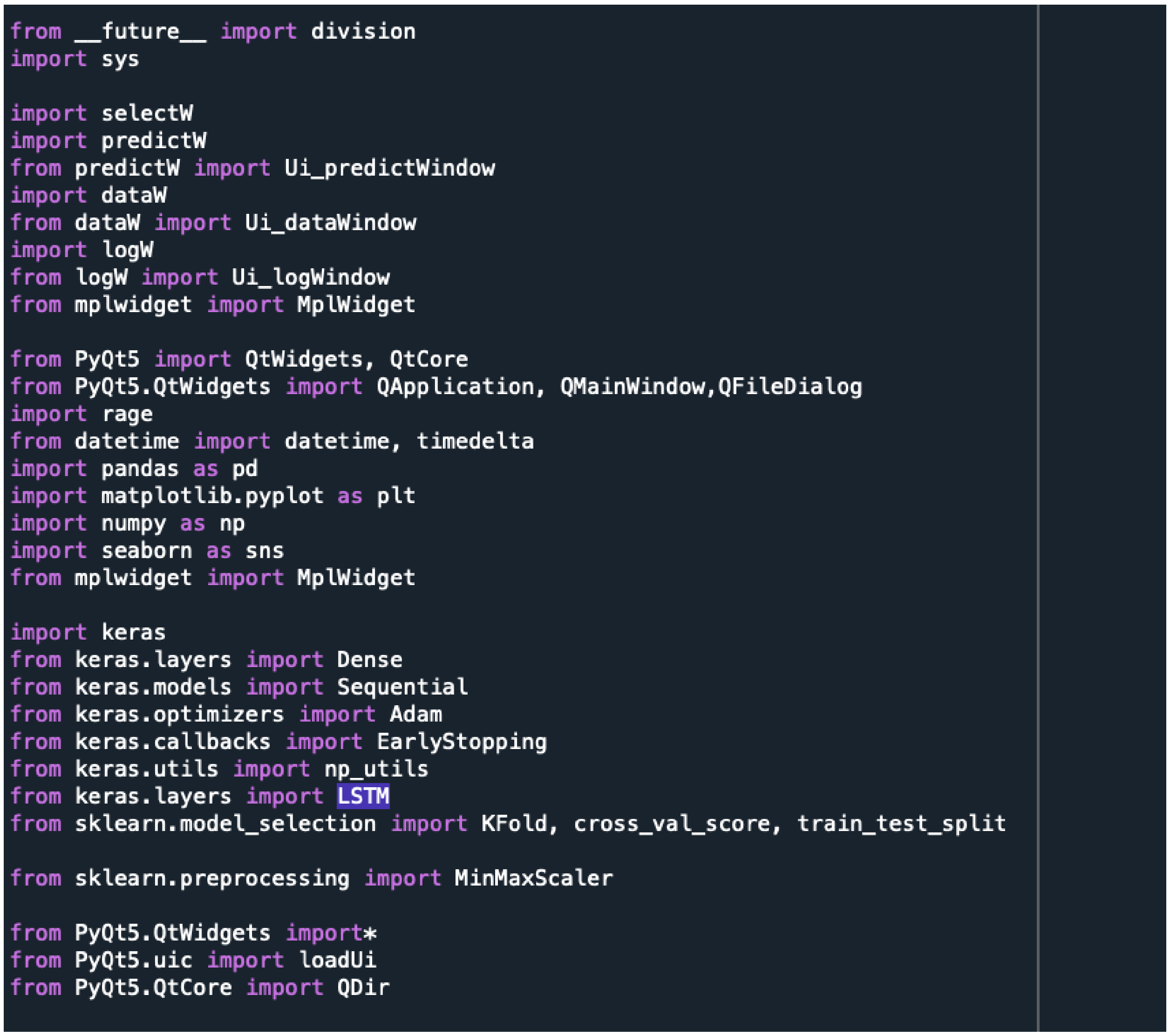
Criterion C - Development

I used PyQt5 instead of TKinter as it allowed for more interactive GUI modifications which made the GUI much more visually appealing and much easier to manage. PyQt5 comes with a designer that utilises a drag-and-drop interface and an extensive library for creation of an intuitive application.

Modules/Libraries Used



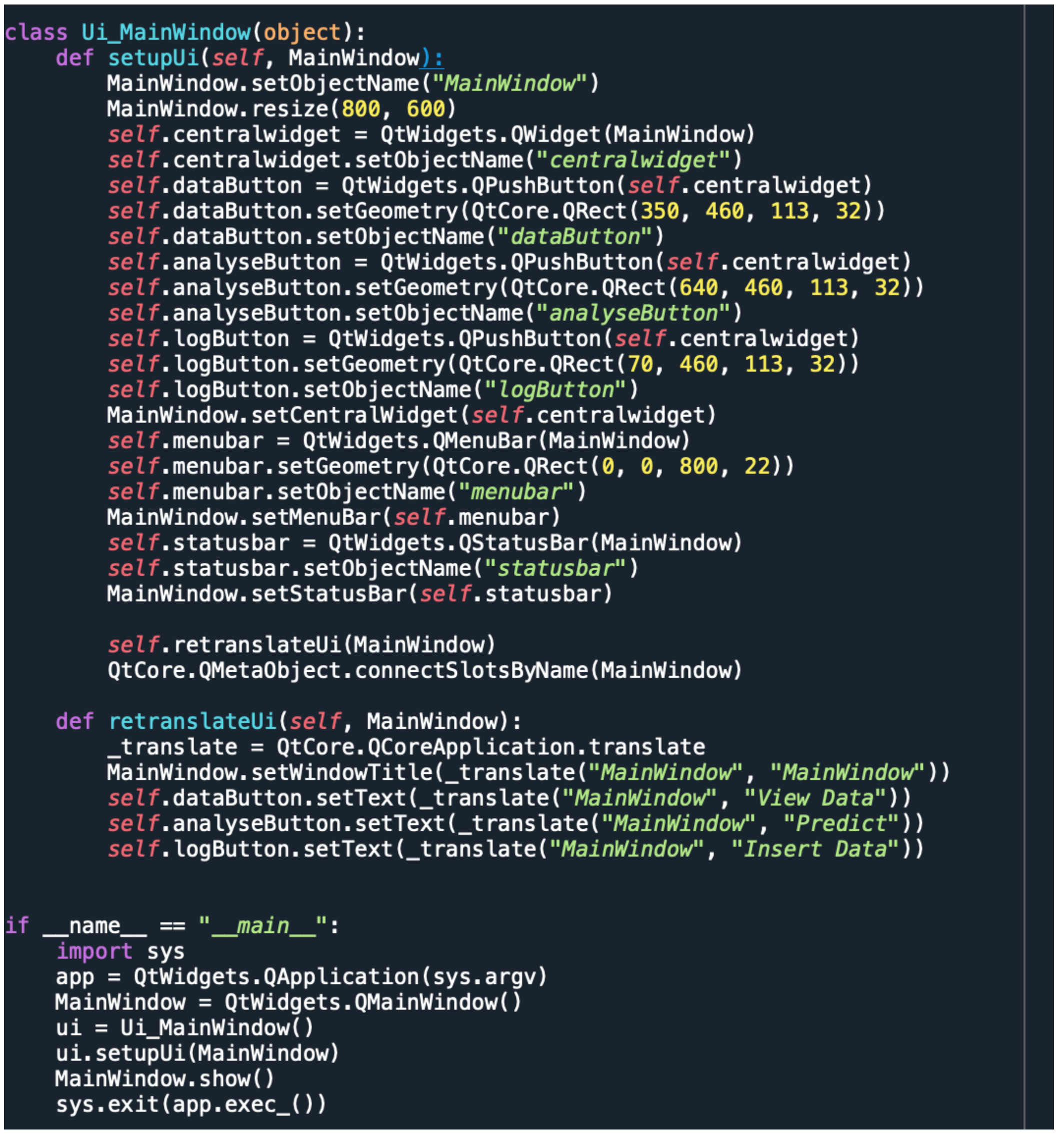
```python
from __future__ import division
import sys

import selectW
import predictW
from predictW import Ui_predictWindow
import dataW
from dataW import Ui_dataWindow
import logW
from logW import Ui_logWindow
from mplwidget import MplWidget

from PyQt5 import QtWidgets, QtCore
from PyQt5.QtWidgets import QApplication, QMainWindow,QFileDialog
import rage
from datetime import datetime, timedelta
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from mplwidget import MplWidget

import keras
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
from keras.utils import np_utils
from keras.layers import LSTM
from sklearn.model_selection import KFold, cross_val_score, train_test_split

from sklearn.preprocessing import MinMaxScaler

from PyQt5.QtWidgets import*
from PyQt5.uic import loadUi
from PyQt5.QtCore import QDir
```

The program is a time series forecast algorithm which requires readable dating format and numerical calculations which is the purpose of the pandas, matplotlib and numpy libraries. These libraries are used to create graphs, tables and perform format conversions.

Keras is a popular machine learning library that offers a wide variety of tools and modules for training AI programs. In this program, forecast is achieved through training the data using a Long Short Term Memory model. This required a number of modules commonly found in ML algorithms.

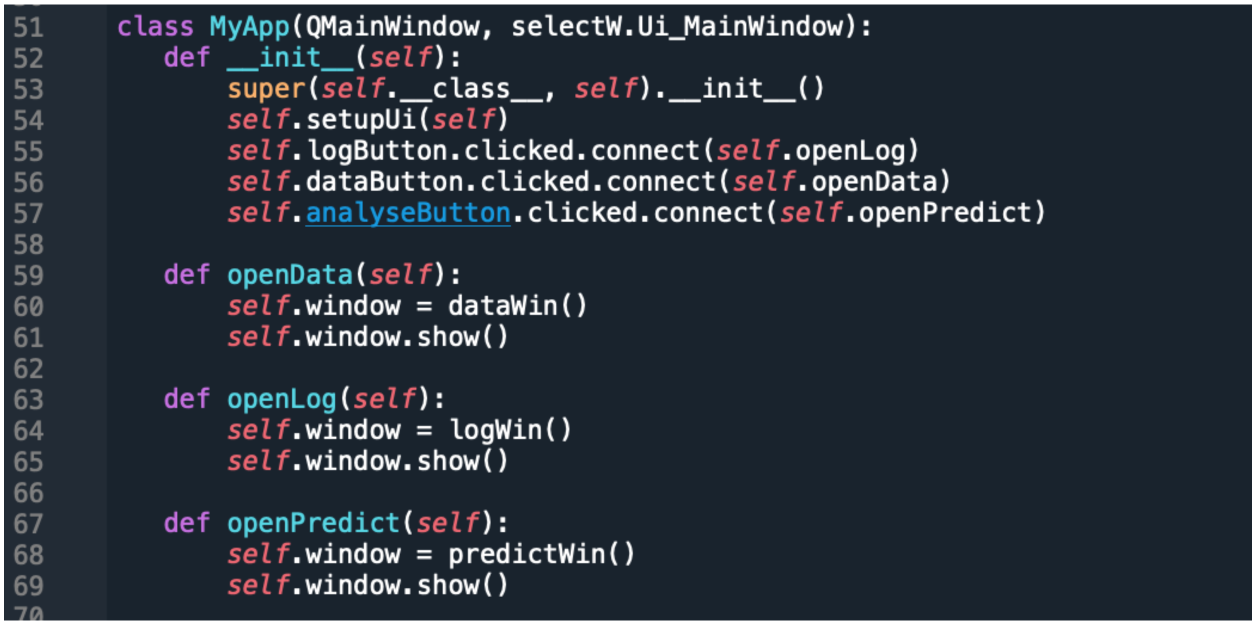
This program heavily relies on encapsulation to manage the usability features. The GUI is modified through exporting the UI files for each of the windows to a python file, which is then imported into an application framework python file (app.py) that customises the code and handles the usability of the gui itself. The app.py file exists when the converted UI program is modified via the python file, the modified python gui file can't be re-converted into a UI file and the added changes are lost if the window is modified in PyQt Designer.

The following code shows an example of the PyQt designer code converted from a UI file for the selectWindow:



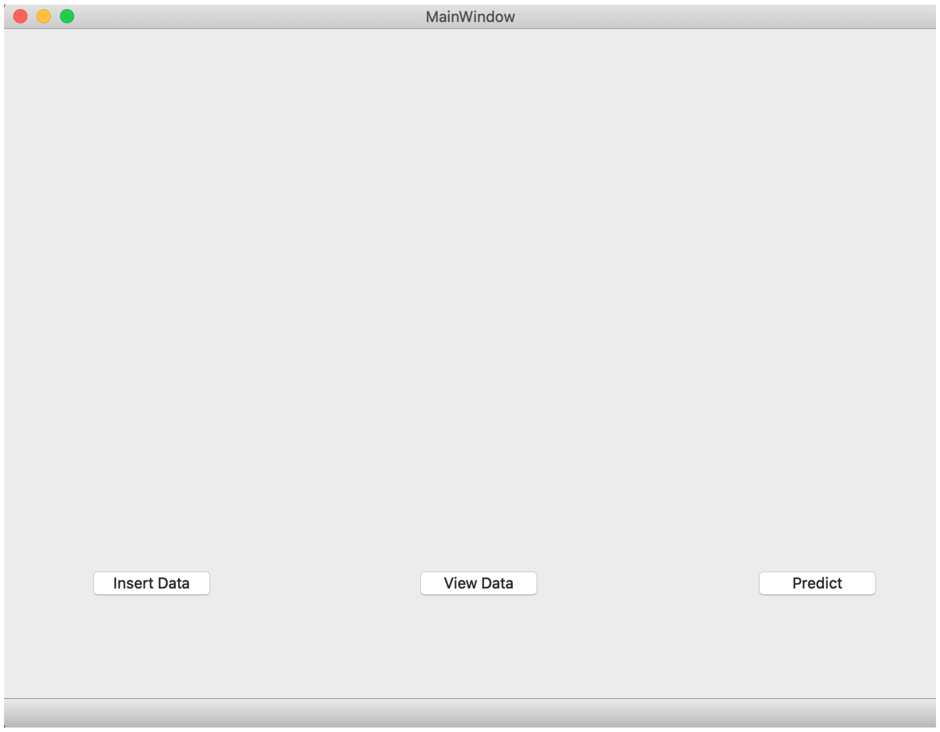
```python
class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(800, 600)
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        self.dataButton = QtWidgets.QPushButton(self.centralwidget)
        self.dataButton.setGeometry(QtCore.QRect(350, 460, 113, 32))
        self.dataButton.setObjectName("dataButton")
        self.analyseButton = QtWidgets.QPushButton(self.centralwidget)
        self.analyseButton.setGeometry(QtCore.QRect(640, 460, 113, 32))
        self.analyseButton.setObjectName("analyseButton")
        self.logButton = QtWidgets.QPushButton(self.centralwidget)
        self.logButton.setGeometry(QtCore.QRect(70, 460, 113, 32))
        self.logButton.setObjectName("logButton")
        MainWindow.setCentralWidget(self.centralwidget)
        self.menubar = QtWidgets.QMenuBar(MainWindow)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 800, 22))
        self.menubar.setObjectName("menubar")
        MainWindow.setMenuBar(self.menubar)
        self.statusbar = QtWidgets.QStatusBar(MainWindow)
        self.statusbar.setObjectName("statusbar")
        MainWindow.setStatusBar(self.statusbar)

        self.retranslateUi(MainWindow)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)

    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
        self.dataButton.setText(_translate("MainWindow", "View Data"))
        self.analyseButton.setText(_translate("MainWindow", "Predict"))
        self.logButton.setText(_translate("MainWindow", "Insert Data"))


if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
    ui = Ui_MainWindow()
    ui.setupUi(MainWindow)
    MainWindow.show()
    sys.exit(app.exec_())
```

This is then imported into the app.py file and the GUI elements are modified and customised as the following displays


```
class MyApp(QMainWindow, selectW.Ui_MainWindow):
    def __init__(self):
        super(self.__class__, self).__init__()
        self.setupUi(self)
        self.logButton.clicked.connect(self.openLog)
        self.dataButton.clicked.connect(self.openData)
        self.analyseButton.clicked.connect(self.openPredict)

    def openData(self):
        self.window = dataWin()
        self.window.show()

    def openLog(self):
        self.window = logWin()
        self.window.show()

    def openPredict(self):
        self.window = predictWin()
        self.window.show()
```

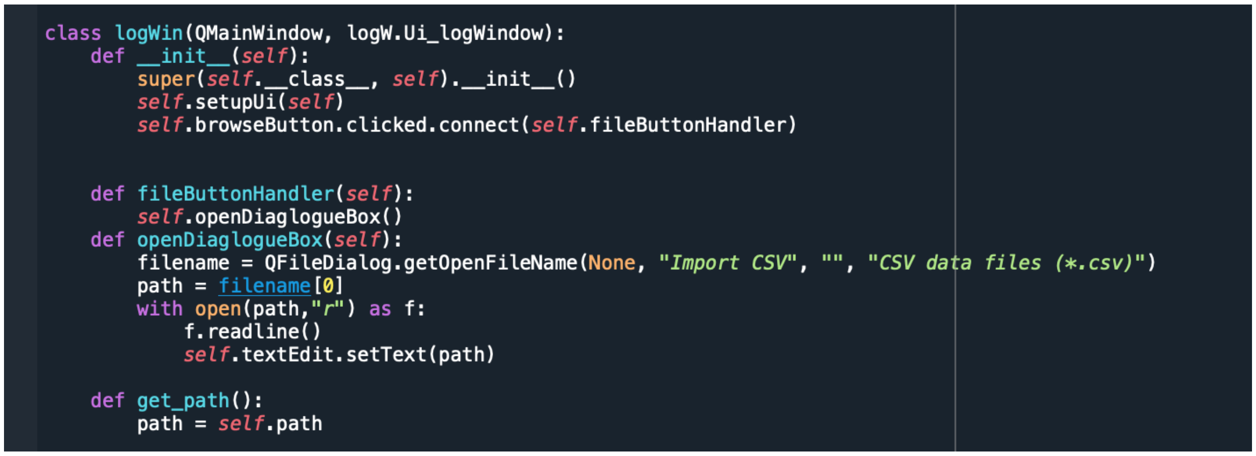
The class inherits the UI layout from it's parent function (being the selectW.Ui_MainWindow) then usability features are added. Here, the activity of the buttons are being linked to opening new windows. Keeping the code in the app.py file ensures that the functionality of each GUI element isn't lost when any of the UI python files is updated.

When the main script is run, it displays the following window



Logging Files

The insert data file allows for data to be manually inserted into the program. The code here utilises modules from the panda library in order to read the file into the program.



```python
class logWin(QMainWindow, logW.Ui_logWindow):
    def __init__(self):
        super(self.__class__, self).__init__()
        self.setupUi(self)
        self.browseButton.clicked.connect(self.fileButtonHandler)


    def fileButtonHandler(self):
        self.openDiaglogueBox()
    def openDiaglogueBox(self):
        filename = QFileDialog.getOpenFileName(None, "Import CSV", "", "CSV data files (*.csv)")
        path = filename[0]
        with open(path,"r") as f:
            f.readline()
            self.textEdit.setText(path)

    def get_path():
        path = self.path
```

The button is connected to a PyQt module called ‘QFileDialog’ which prompts open a file browser and reads the selected file into ‘filename’. The arguments limit the browser to only accepting CSV files since that’s what the program is limited to working with (as the prediction algorithm only works with csv files) and to prevent any errors from uploading an unreadable file.

The class also has the function get_path() because the file uploaded is the one that will be worked with in the data visualisation window (dataWindow) and the predict window thereby requiring a function which gives the other classes access to the uploaded data.

The logging window’s GUI is as follows

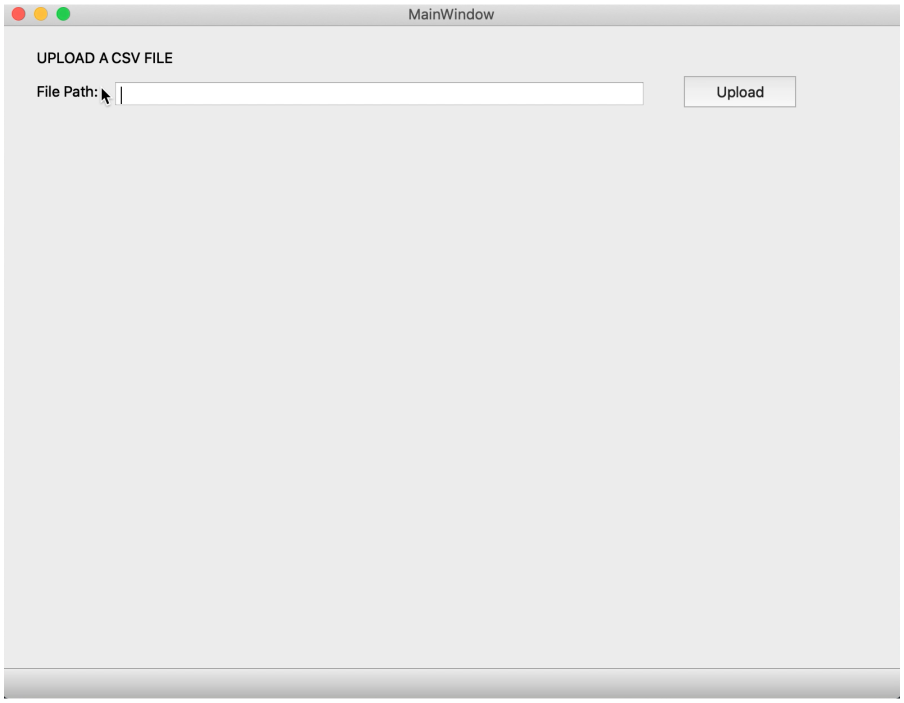
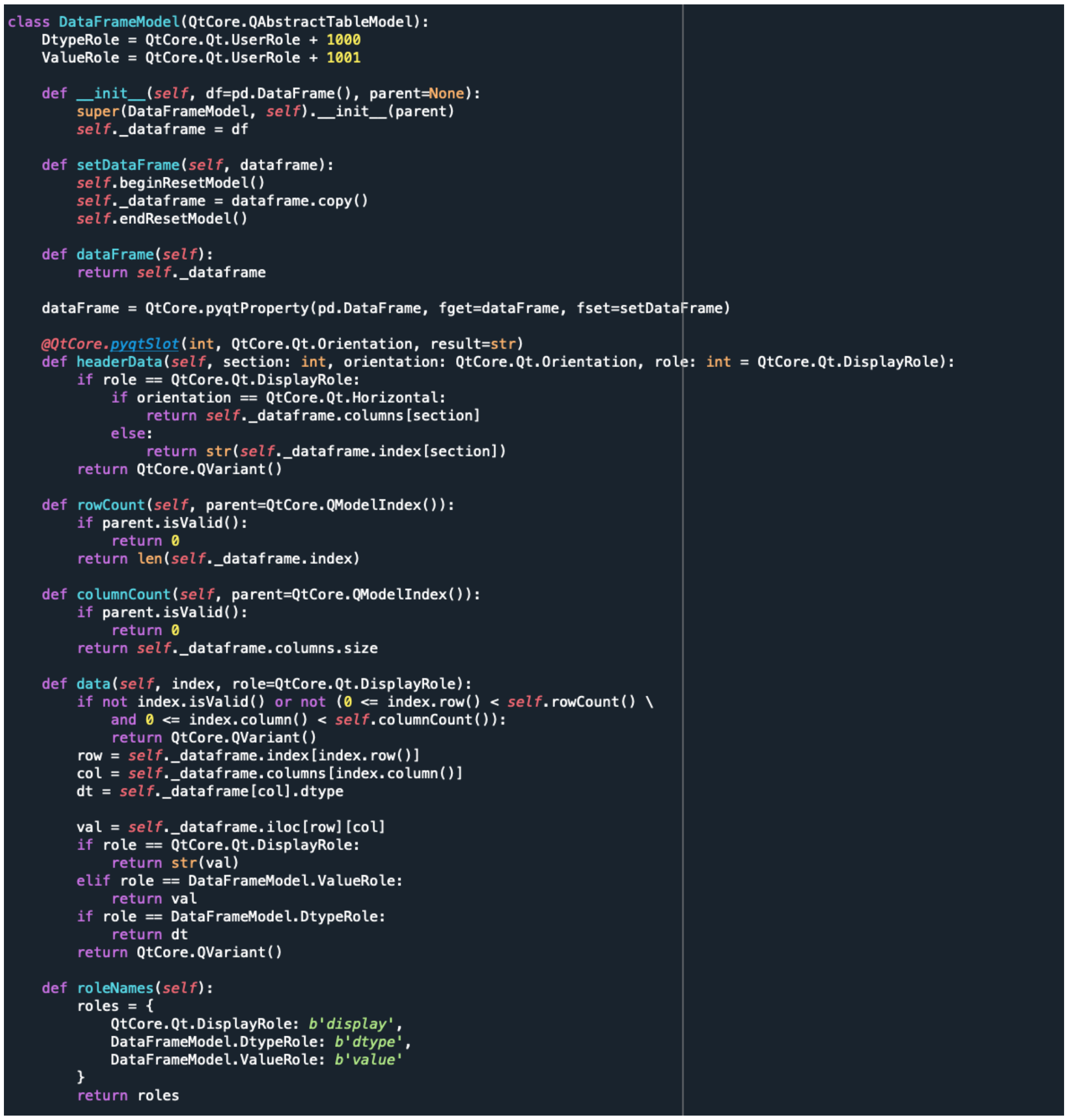


Table Display

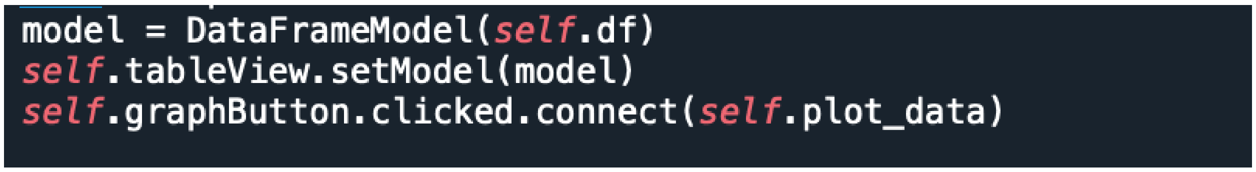
The data frames of this application are read using modules from the python pandas library and are given in a 2D array. To embed these tables into the application, a new model had to be created to display the data through the PyQt Table widgets.



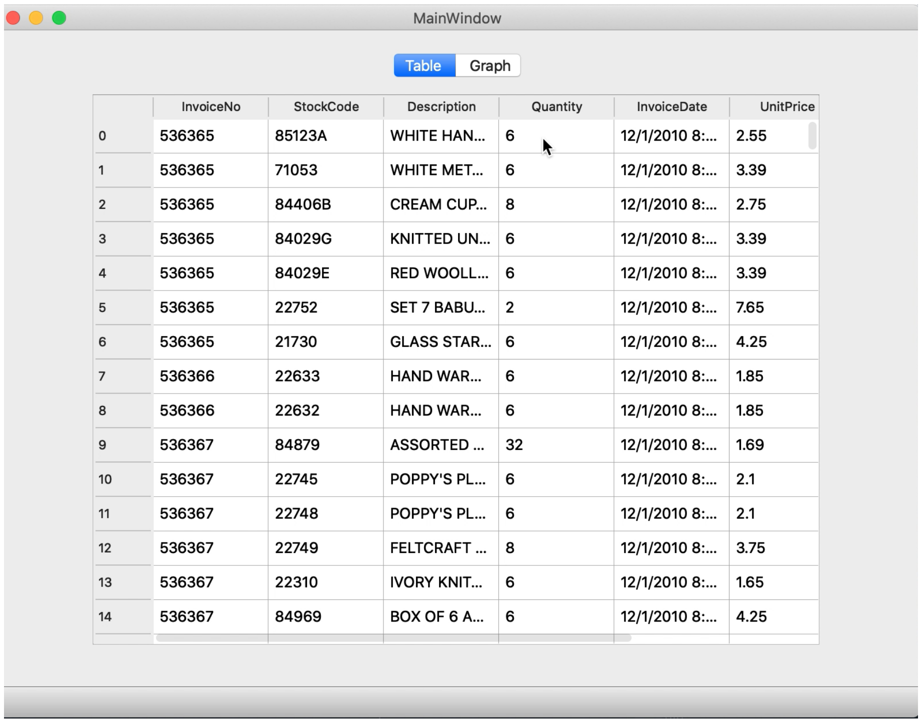
```python
class DataFrameModel(QtCore.QAbstractTableModel):
    DtypeRole = QtCore.Qt.UserRole + 1000
    ValueRole = QtCore.Qt.UserRole + 1001

    def __init__(self, df=pd.DataFrame(), parent=None):
        super(DataFrameModel, self).__init__(parent)
        self._dataframe = df

    def setDataFrame(self, dataframe):
        self.beginResetModel()
        self._dataframe = dataframe.copy()
        self.endResetModel()

    def dataFrame(self):
        return self._dataframe

    dataFrame = QtCore.pyqtProperty(pd.DataFrame, fget=dataFrame, fset=setDataFrame)

    @QtCore.pyqtSlot(int, QtCore.Qt.Orientation, result=str)
    def headerData(self, section: int, orientation: QtCore.Qt.Orientation, role: int = QtCore.Qt.DisplayRole):
        if role == QtCore.Qt.DisplayRole:
            if orientation == QtCore.Qt.Horizontal:
                return self._dataframe.columns[section]
            else:
                return str(self._dataframe.index[section])
        return QtCore.QVariant()

    def rowCount(self, parent=QtCore.QModelIndex()):
        if parent.isValid():
            return 0
        return len(self._dataframe.index)

    def columnCount(self, parent=QtCore.QModelIndex()):
        if parent.isValid():
            return 0
        return self._dataframe.columns.size

    def data(self, index, role=QtCore.Qt.DisplayRole):
        if not index.isValid() or not (0 <= index.row() < self.rowCount() \
            and 0 <= index.column() < self.columnCount()):
            return QtCore.QVariant()
        row = self._dataframe.index[index.row()]
        col = self._dataframe.columns[index.column()]
        dt = self._dataframe[col].dtype

        val = self._dataframe.iloc[row][col]
        if role == QtCore.Qt.DisplayRole:
            return str(val)
        elif role == DataFrameModel.ValueRole:
            return val
        if role == DataFrameModel.DtypeRole:
            return dt
        return QtCore.QVariant()

    def roleNames(self):
        roles = {
            QtCore.Qt.DisplayRole: b'display',
            DataFrameModel.DtypeRole: b'dtype',
            DataFrameModel.ValueRole: b'value'
        }
        return roles
```

The most important method there is the data method, which does the conversions to make the data frame format embeddable inside a PyQt program.

This is then called in the main function of the loop to display the table.



```python
model = DataFrameModel(self.df)
self.tableView.setModel(model)
self.graphButton.clicked.connect(self.plot_data)
```

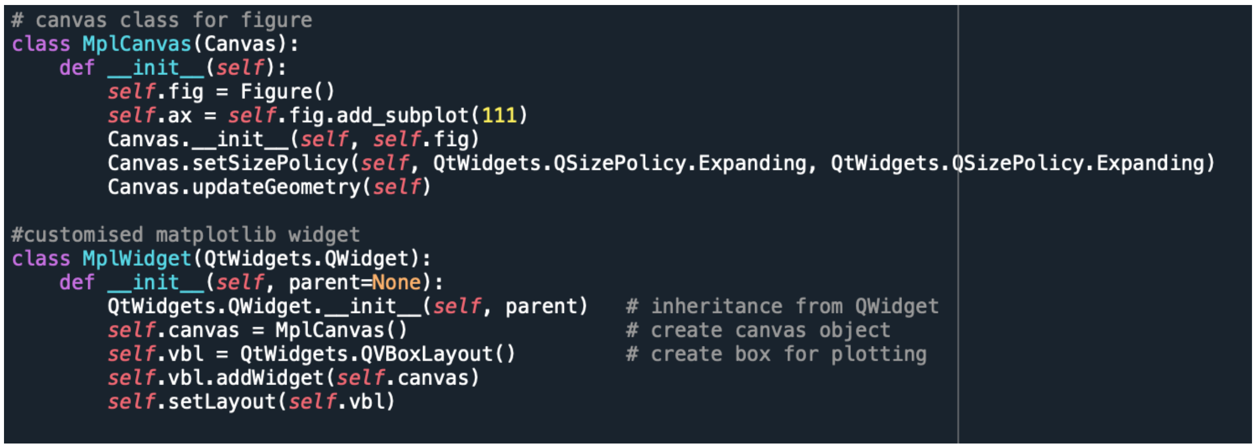


	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice
0	536365	85123A	WHITE HAN...	6	12/1/2010 8:...	2.55
1	536365	71053	WHITE MET...	6	12/1/2010 8:...	3.39
2	536365	84406B	CREAM CUP...	8	12/1/2010 8:...	2.75
3	536365	84029G	KNITTED UN...	6	12/1/2010 8:...	3.39
4	536365	84029E	RED WOOLL...	6	12/1/2010 8:...	3.39
5	536365	22752	SET 7 BABU...	2	12/1/2010 8:...	7.65
6	536365	21730	GLASS STAR...	6	12/1/2010 8:...	4.25
7	536366	22633	HAND WAR...	6	12/1/2010 8:...	1.85
8	536366	22632	HAND WAR...	6	12/1/2010 8:...	1.85
9	536367	84879	ASSORTED ...	32	12/1/2010 8:...	1.69
10	536367	22745	POPPY'S PL...	6	12/1/2010 8:...	2.1
11	536367	22748	POPPY'S PL...	6	12/1/2010 8:...	2.1
12	536367	22749	FELTCRAFT ...	8	12/1/2010 8:...	3.75
13	536367	22310	IVORY KNIT...	6	12/1/2010 8:...	1.65
14	536367	84969	BOX OF 6 A...	6	12/1/2010 8:...	4.25

Graphing

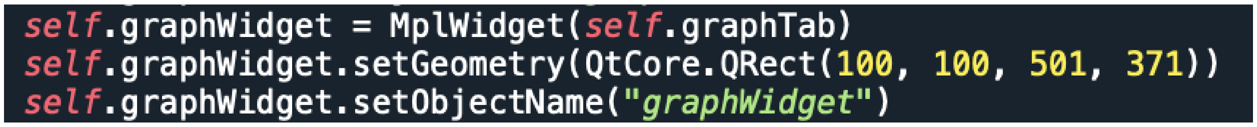
An important aspect of this program was generating graphs for forecasting, which had to be done through embedding matplotlib plots (which are a python library used for data visualisation). Matplotlib is much more versatile in terms of data plotting and it creates graphs that are much cleaner and easy to read as well as modify in code.

The graph is embedded in a widget that was customised to be compatible with matplotlib graphs through the code below:



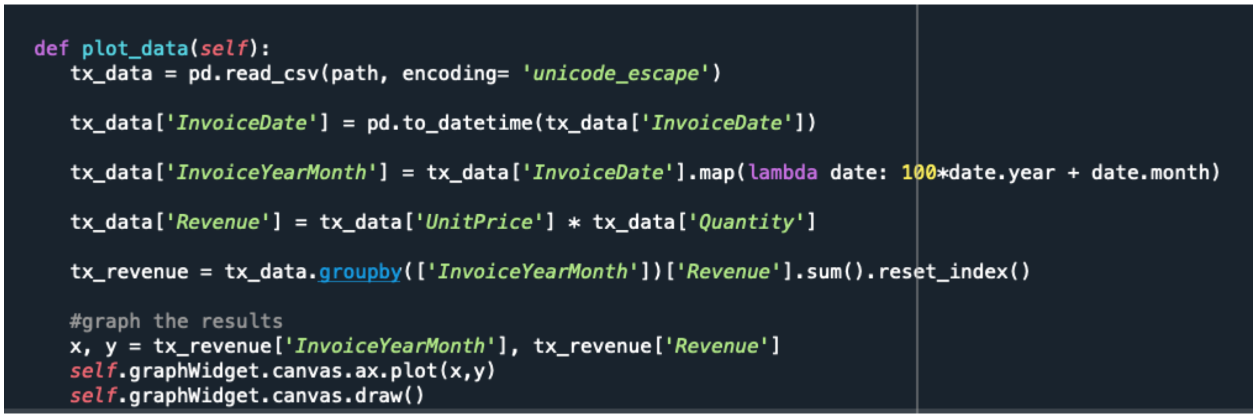
```python
# canvas class for figure
class MplCanvas(Canvas):
    def __init__(self):
        self.fig = Figure()
        self.ax = self.fig.add_subplot(111)
        Canvas.__init__(self, self.fig)
        Canvas.setSizePolicy(self, QtWidgets.QSizePolicy.Expanding, QtWidgets.QSizePolicy.Expanding)
        Canvas.updateGeometry(self)

#customised matplotlib widget
class MplWidget(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)    # inheritance from QWidget
        self.canvas = MplCanvas()                   # create canvas object
        self.vbl = QtWidgets.QVBoxLayout()          # create box for plotting
        self.vbl.addWidget(self.canvas)
        self.setLayout(self.vbl)
```

The custom widget is applied to the UI code which allows it to plot matplotlib graphs.



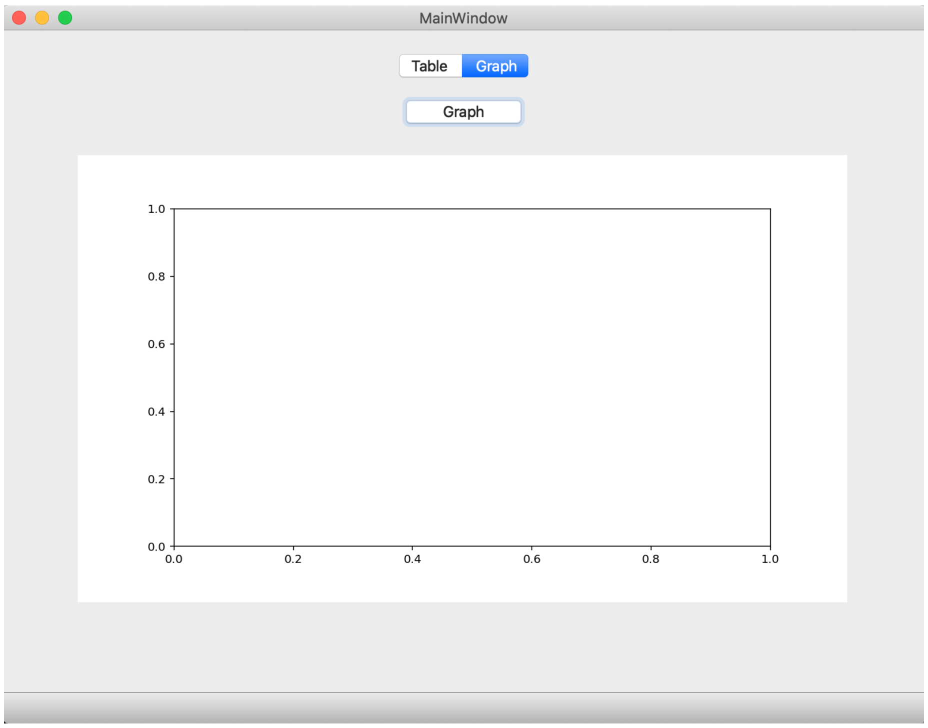
```python
self.graphWidget = MplWidget(self.graphTab)
self.graphWidget.setGeometry(QtCore.QRect(100, 100, 501, 371))
self.graphWidget.setObjectName("graphWidget")
```

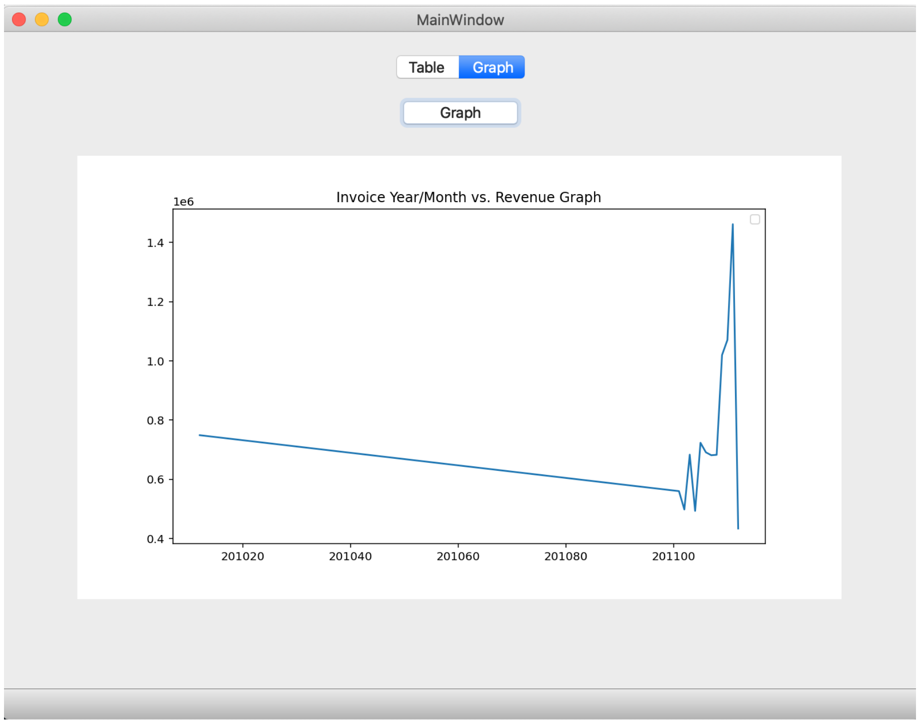
The MplCanvas is then called in the app.py code for the data window to display the graph through putting it in the widget.



```python
def plot_data(self):
    tx_data = pd.read_csv(path, encoding= 'unicode_escape')

    tx_data['InvoiceDate'] = pd.to_datetime(tx_data['InvoiceDate'])

    tx_data['InvoiceYearMonth'] = tx_data['InvoiceDate'].map(lambda date: 100*date.year + date.month)

    tx_data['Revenue'] = tx_data['UnitPrice'] * tx_data['Quantity']

    tx_revenue = tx_data.groupby(['InvoiceYearMonth'])['Revenue'].sum().reset_index()

    #graph the results
    x, y = tx_revenue['InvoiceYearMonth'], tx_revenue['Revenue']
    self.graphWidget.canvas.ax.plot(x,y)
    self.graphWidget.canvas.draw()
```

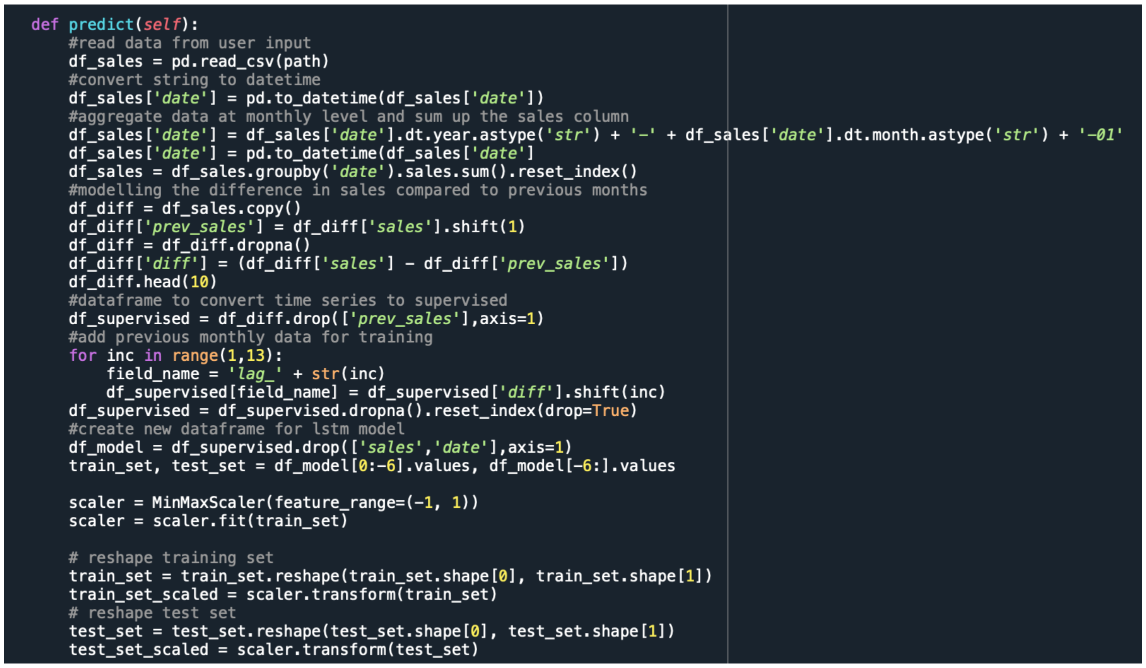
Following is what is originally displayed in the graph upon being prompted open, and what is displayed after button is clicked.





Predictive Algorithm

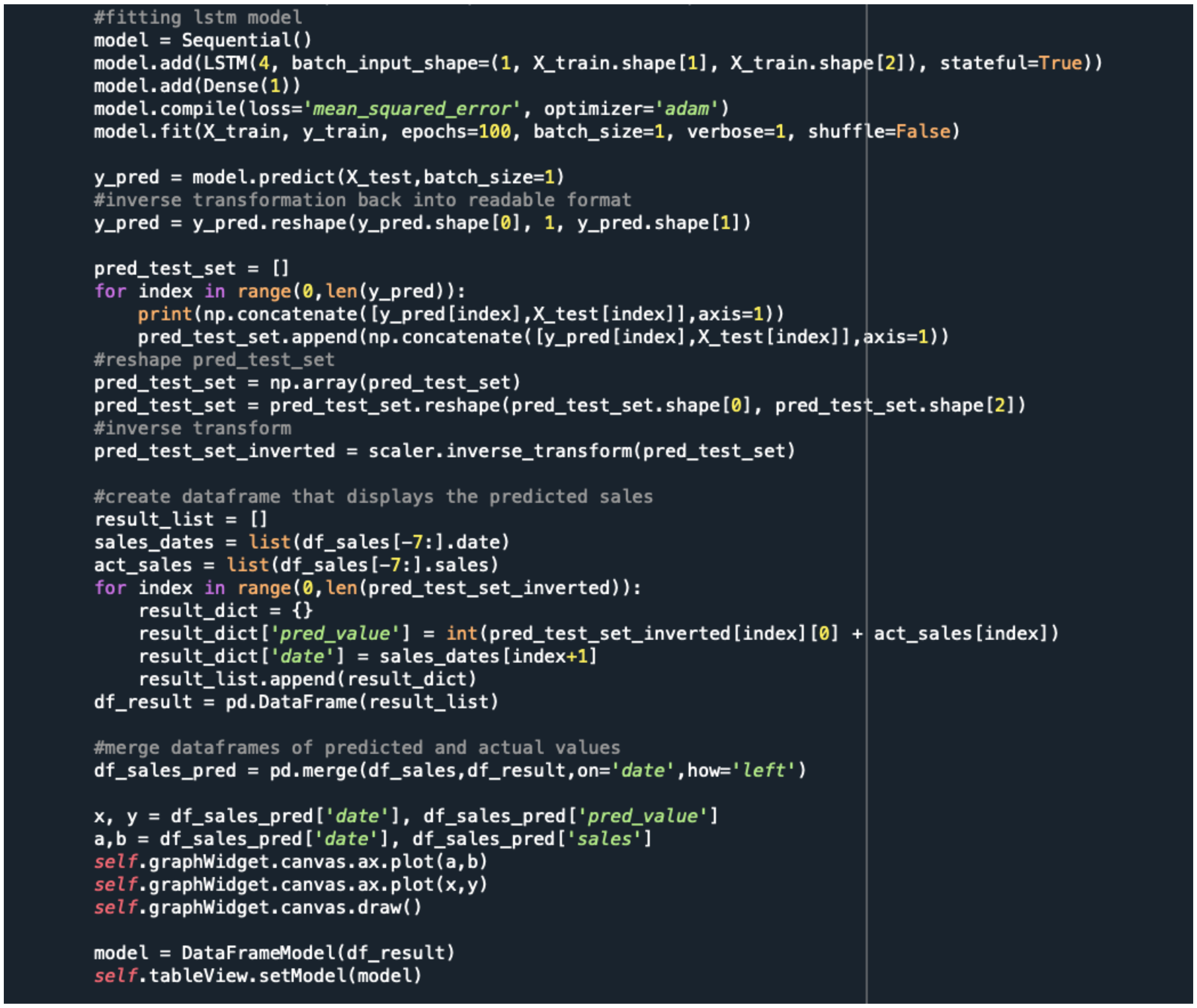
The predictive algorithm takes in the dataset and takes in the dates and sales and creates a new dataframe. Some pre-processing has to be completed before the algorithm could actually fit the data into the machine learning model.



```
def predict(self):
    #read data from user input
    df_sales = pd.read_csv(path)
    #convert string to datetime
    df_sales['date'] = pd.to_datetime(df_sales['date'])
    #aggregate data at monthly level and sum up the sales column
    df_sales['date'] = df_sales['date'].dt.year.astype('str') + '-' + df_sales['date'].dt.month.astype('str') + '-01'
    df_sales['date'] = pd.to_datetime(df_sales['date']
    df_sales = df_sales.groupby('date').sales.sum().reset_index()
    #modelling the difference in sales compared to previous months
    df_diff = df_sales.copy()
    df_diff['prev_sales'] = df_diff['sales'].shift(1)
    df_diff = df_diff.dropna()
    df_diff['diff'] = (df_diff['sales'] - df_diff['prev_sales'])
    df_diff.head(10)
    #dataframe to convert time series to supervised
    df_supervised = df_diff.drop(['prev_sales'],axis=1)
    #add previous monthly data for training
    for inc in range(1,13):
        field_name = 'lag_' + str(inc)
        df_supervised[field_name] = df_supervised['diff'].shift(inc)
    df_supervised = df_supervised.dropna().reset_index(drop=True)
    #create new dataframe for lstm model
    df_model = df_supervised.drop(['sales','date'],axis=1)
    train_set, test_set = df_model[0:-6].values, df_model[-6:].values

    scaler = MinMaxScaler(feature_range=(-1, 1))
    scaler = scaler.fit(train_set)

    # reshape training set
    train_set = train_set.reshape(train_set.shape[0], train_set.shape[1])
    train_set_scaled = scaler.transform(train_set)
    # reshape test set
    test_set = test_set.reshape(test_set.shape[0], test_set.shape[1])
    test_set_scaled = scaler.transform(test_set)
```

The data is processed and split into a 'train' and 'test' set that will be sent through the machine learning algorithm, which is a standard practice when working with such algorithms.

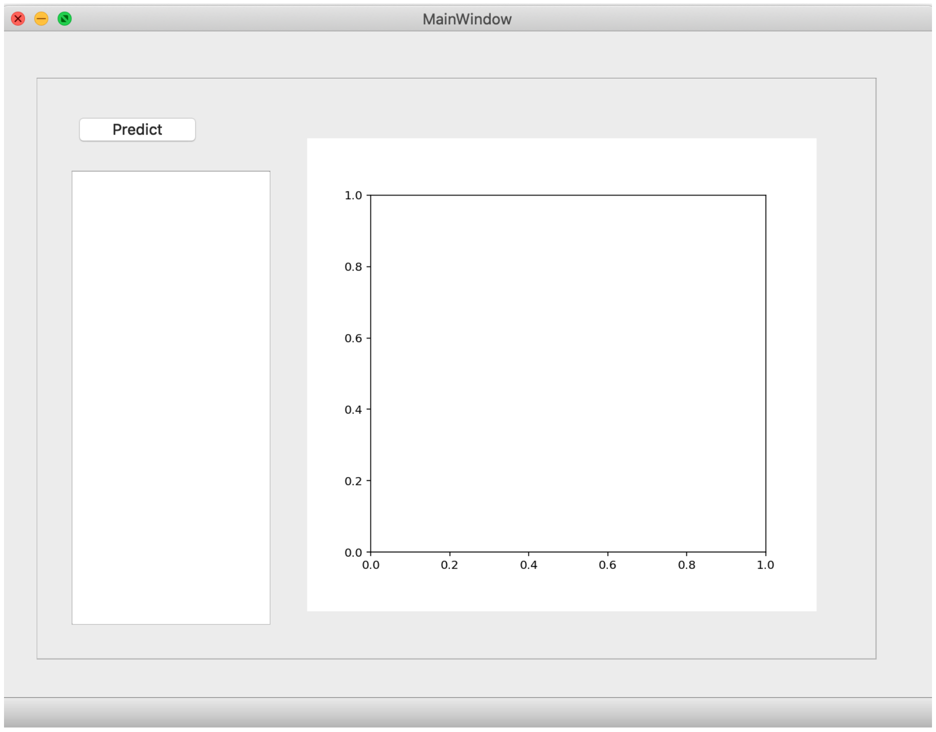
The algorithm uses a Long-Short-Term-Memory neural network model, which is a model that uses previous data to continue improving the forecasting ability in a sequential manner.

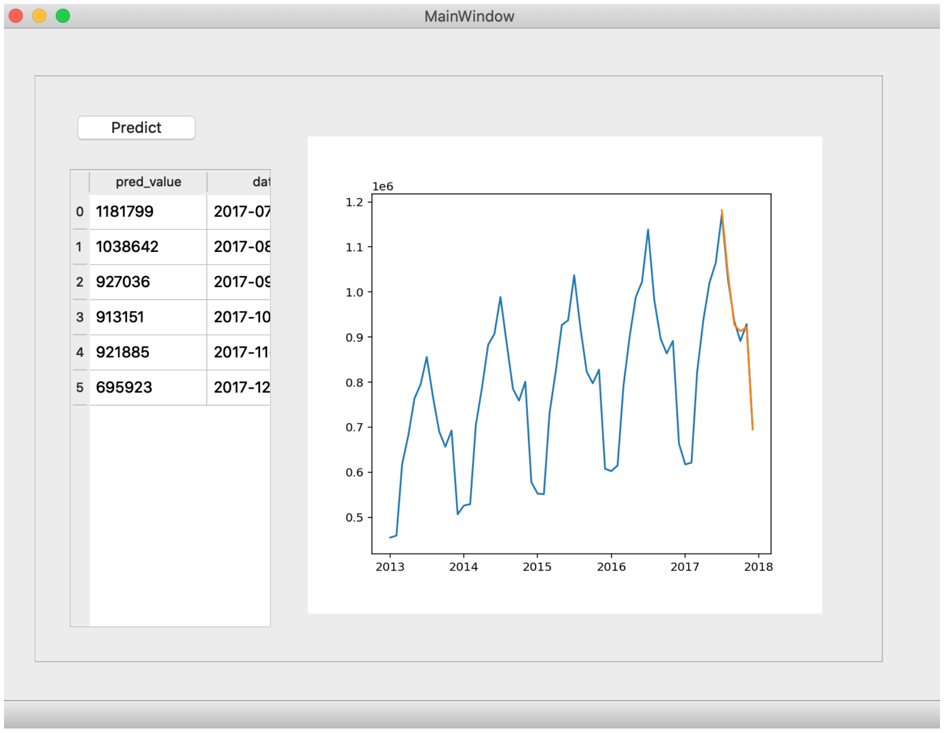


```
        #fitting lstm model
        model = Sequential()
        model.add(LSTM(4, batch_input_shape=(1, X_train.shape[1], X_train.shape[2]), stateful=True))
        model.add(Dense(1))
        model.compile(loss='mean_squared_error', optimizer='adam')
        model.fit(X_train, y_train, epochs=100, batch_size=1, verbose=1, shuffle=False)

        y_pred = model.predict(X_test,batch_size=1)
        #inverse transformation back into readable format
        y_pred = y_pred.reshape(y_pred.shape[0], 1, y_pred.shape[1])

        pred_test_set = []
        for index in range(0,len(y_pred)):
            print(np.concatenate([y_pred[index],X_test[index]],axis=1))
            pred_test_set.append(np.concatenate([y_pred[index],X_test[index]],axis=1))
        #reshape pred_test_set
        pred_test_set = np.array(pred_test_set)
        pred_test_set = pred_test_set.reshape(pred_test_set.shape[0], pred_test_set.shape[2])
        #inverse transform
        pred_test_set_inverted = scaler.inverse_transform(pred_test_set)

        #create dataframe that displays the predicted sales
        result_list = []
        sales_dates = list(df_sales[-7:].date)
        act_sales = list(df_sales[-7:].sales)
        for index in range(0,len(pred_test_set_inverted)):
            result_dict = {}
            result_dict['pred_value'] = int(pred_test_set_inverted[index][0] + act_sales[index])
            result_dict['date'] = sales_dates[index+1]
            result_list.append(result_dict)
        df_result = pd.DataFrame(result_list)

        #merge dataframes of predicted and actual values
        df_sales_pred = pd.merge(df_sales,df_result,on='date',how='left')

        x, y = df_sales_pred['date'], df_sales_pred['pred_value']
        a,b = df_sales_pred['date'], df_sales_pred['sales']
        self.graphWidget.canvas.ax.plot(a,b)
        self.graphWidget.canvas.ax.plot(x,y)
        self.graphWidget.canvas.draw()

        model = DataFrameModel(df_result)
        self.tableView.setModel(model)
```

The algorithm takes around a minute to generate results, which is relatively fast considering the program is being run through a non-GPU supported environment which highlights the benefits of using python in place of other languages for implementing extensive data analysis algorithms. The results are as shown in the window below.

The sale forecast feature of this program is the most crucial component as it is what the program was specifically designed for.





	pred_value	dat
0	1181799	2017-07
1	1038642	2017-08
2	927036	2017-09
3	913151	2017-10
4	921885	2017-11
5	695923	2017-12