Data_gen_mk2.py

Purpose:

The goal of the data generator was to have a program that could take in a text file with items and their attributes and generate used store items based on various other factors. Those other factors are the day of the week, deviation from a rolling 30 weather average, levels of precipitation, and major commercial calendar holidays. The result is a csv file with markings on when an item is ordered, how much, and assigns an order number to it.

Libraries used:

```
import random
import requests
import os
from datetime import datetime, date, timedelta
import math
import weather_input
import copy
from xlwt import Workbook
import csv
import pandas as pd
import glob
import csvsplitter
```

- Import random – This simply used to generate pseudo-random numbers. In the data gen, it is used to randomize an items initial average use, so no two runs have the same results. https://docs.python.org/3/library/random.html
- Import requests – Allows for simple http requests. This was used in downloading the latest list changes from my github. That way any member at any time can run the data gen without issue. https://docs.python-requests.org/en/latest/
- Import os – Allows for the use of operating system dependent functionality. This was used in a couple ways. First, this was used to the location of where the program was saved to. This way we could tailor the file output to be the same no matter who used it. Second was for system calls. The last two things the_loop does is change the current directory to where the machine learning is and the next is that it runs the code. https://docs.python.org/3/library/os.html
- From datetime import datetime, date, timedelta – To separate when files where generated, we decided to use a time date naming scheme to indicate different runs. This way we have a history that we can call back to. That was the use of datetime. For date and timedelta, these were used in the conversion from integer to actual date within a year. We used this when creating the 6 years of data for item purchases. https://docs.python.org/3/library/datetime.html
- Import math – A handy module that was really only used for its floor and ceiling operations. https://docs.python.org/3/library/math.html
- Import weather_input – This is the program that takes in weather data and applies values for the data gen to work around.
- Import copy – This was used for its deepcopy functionality. When we were copying a list from weather_input to data_gen_mk2, changes to the list in data_gen_mk2 would be seen in

weather_input. To correct for this, we deep copied the list to make it a separate entity. https://docs.python.org/3/library/copy.html

- From xlwt import workbook – An old library that handles Microsoft excel files versions 95 to 2003. This library is no longer supported and recommends using openpyxl instead. For our purposes though, we are able to open, save, and create excel files. https://pypi.org/project/xlwt/
- Import pandas as pd – A powerful data analysis tool. The use of it in the data generation tool was to convert xls to csv and merge many xls files into on large file. We needed to merge the files as each year is recorded separately. For simplicity, we then merged once they were all generated and then converted to a csv. https://pandas.pydata.org/docs/user_guide/10min.html
- import glob - This was used in the merging process to grab all the files matching a certain description into a list. This way, when all are read, they can then be merged. https://docs.python.org/3/library/glob.html
- import csvsplitter – This is another program aid the data generator. This takes the merged csv and then splits it up into individual items.

Classes:

There are two classes used, item and order. First, item.

```
class item:
    def __init__(self, name, min, max, deviation, id, ival):
        self.name = name
        self.min = min
        self.max = max
        self.dev = deviation
        self.orders = []
        self.id = id
        self.init_val = ival
        self.store_avg = 0
        self.shipped = False
        self.shipping_time = 2
        self.two_week_grace = 0
        self.holiday_mod = 2
        self.weekend_mod = 0
        self.rolling_avg = []
```

When an item is created, six attributes are taken in. Name, min, max, deviation, id, and ival.

- Name – This is the description of the item.
- Min – Minimum value that the item can be used initially.
- Max – Maximum value that the item can be used initially.
- Deviation – Intention was for this to be used to create chaos into the items use. However, this was never implemented as we used other means to introduce controlled chaos. Such as weather, holidays, etc. You can set this to 0 if adding new items to the item text file.
- Id – The identification number of the item.
- Ival – The stock value for the item based on the average of the min and max values multiplied by the time period in which an item is ordered for. This is discussed later in fill_list().

The rest of the values.

- Self.orders = [] – This tracks when an item is ordered.
- Self.store_avg = The stores average use for an item. This value is chosen randomly from the values between min and max.
- Self.shipped – Used in tracking when an item is ordered.
- Self.shipping_time – Tracks time it takes for an order to arrive once ordered.
- Self.two_weeks_grace – The period for which the store orders enough products for. Initially this was two weeks, but we changed it to a week to generate more data.
- Self.holiday_mod – A flat value that helps denote a stores extra traffic during a holiday. So, the 2 represents items use multiplied by two. 2 is an arbitrary value as we do not have actual data for foot traffic in the stores during holidays.
- Self.rolling_avg – A list that tracks the previous *two_weeks* days of item use. Used in ordering supplies to the store. This gets multiplied by two_weeks_grace to order a quantity that should cover the store for a week. Or for however long the value gets changed to.

```
class order:
    def __init__(self, name, day, qty, num):
        self.item_name = name
        self.day = day
        self.qty = qty
        self.order_num = 0
        self.item_num = num
```

The order class was used as a way of attributing order numbers to the items ordered. It is also the only attributes that get written to the csv and xls files. Four attributes are given to an order. The items name, day, qty, and num.

- Name – Same as item.
- Day – The day on which the item is.
- Qty – How many items where ordered.
- Num – The items id.
- Self.order_num – Initially set to 0 but gets an order number attached later.

Global variables:

```
#init the obj array for the items
order_list = []
item_list = []
two_weeks = 7
holiday = [1, 45, 76, 104, 129, 171, 213, 214, 215, 216, 217, 218, 219,
           220, 221, 222, 223, 224, 225, 226, 227, 228, 229 , 230, 304,
           331, 338, 345, 352, 353, 355, 356, 357, 360]
#this will pull the most current list from my git
url = "https://raw.githubusercontent.com/jgthornb/files/main/item_generation.txt"
page = requests.get(url)
local_file = 'item_copy.txt'
num_order = 0
xlsPos = 0
#saves a copy to the current folder. can be deleted later.
with open(local_file, 'wb') as file:
    file.write(page.content)
#read the file and put the obj in to the array
fname = os.getcwd() + '\\' + local_file
```

A few global variables where necessary to facilitate the operation of the data generation. These ranged from lists to downloading a current item pool.

- Order_list – Stores the order class objects.
- Item_list - Stores the item class objects.
- Holiday – This is a list of the most shopped holidays in the United States. The integer values are representations for the day they fall in the year. So, 360 is December 25. Used in determining if to use the holiday mod.
- Code segment from url to fname, excluding num_order and xlsPos – This goes to a github page and downloads the latest item.txt and save it to the computer. Used in fill_list().
- Num_order – Used to keep track of the current order number when they are being applied. This is global to avoid reinitializing every loop.
- xlsPos – Used to keep track of the line in writing to xls.

Operation:

```
#what initiates all the work
def execute():
    global order_list
    fill_list()
    the_loop(1)
    order_list.sort(key=lambda x:x.day)
    apply_order_num()
    #generate a filename that can be used in both xls and csv
    folder = timeDate()
    file = makeFileName(folder)
    xlsToCsv(save_xls(order_list, '2016',file,folder))
    order_list = []
    the_loop(2)
    order_list.sort(key=lambda x:x.day)
    apply_order_num()
    xlsToCsv(save_xls(order_list,'2017',file,folder))
    the_loop(3)
    order_list.sort(key=lambda x:x.day)
    apply_order_num()
    xlsToCsv(save_xls(order_list,'2018',file,folder))
    the_loop(4)
    order_list.sort(key=lambda x:x.day)
    apply_order_num()
    xlsToCsv(save_xls(order_list,'2019',file,folder))
    the_loop(5)
    order_list.sort(key=lambda x:x.day)
    apply_order_num()
    xlsToCsv(save_xls(order_list,'2020',file,folder))
    the_loop(6)
    order_list.sort(key=lambda x:x.day)
    apply_order_num()
    xlsToCsv(save_xls(order_list,'2021',file,folder))
    to_splitter = mergeXls(folder)
    print(to_splitter)
    csvsplitter.seperate(to_splitter)
    os.chdir("..\ML")
    os.system("python MultiItemML.py")
```

While the layout is not the most beautiful, it does operate. There are repetitious function calls that could also be changed, but to do so would require the creation of support functions that would probably take up just as much space. The function named execute() is what makes this tool work. Essentially it could be considered middle management as it does not create anything, only tells others what to do.

- Global order_list – Points to the global variable and noting that we are not creating a local variable.

- Fill_list() - This function takes the downloaded copy of the item list and then fills item_list[] with

```
def fill_list():
    with open(fname, "r") as myfile:
        for line in myfile:
            x = line.strip().split(", ")
            item_list.append(item(x[0], int(x[1]), int(x[2]), int(x[3]), x[4], int(x[5])))
    for i in range(len(item_list)):
        get_item_avg(item_list[i])
        item_list[i].two_week_grace = item_list[i].store_avg * two_weeks
```

the items. Next, in the for loop, another function is called, get_item_avg. This is a simple function that finds the items average use for the store for this run of the program. Next, we find

```
#prints out a specific object
def get_item_avg(self):
    self.store_avg = (random.randrange(self.min, self.max, 1))
```

the *two_week_grace* for the item. Initially this was two weeks, but we needed more data, so we changed the variable *two_weeks* to 7 from 14. This is the initial value for *two_week_grace*, this gets modified later to other value based on rolling average.

- The_loop(year) – This is the function that really does all the number crunching for the data generator. The variable *year* is used to know which year we are running data on. The values for

```
def the_loop(year):
    weather_input.execute(year)
    weather_cpy = copy.deepcopy(weather_input.weather_list)
    for j in range(len(item_list)):
        holiday_incr = 0
        for i in range(365):
            #see if there is a holiday coming up, check for weather event. this is where we will make modifications to the avg
            mod_avg = item_list[j].store_avg * weather_cpy[i].weather_reduction
            if i == holiday[holiday_incr]:
                mod_avg = mod_avg * item_list[j].holiday_mod
                holiday_incr = (holiday_incr + 1) %  len(holiday)
            resupply_stock(item_list[j],i)
            weekend_check(item_list[j], i)
            find_avg(item_list[j], i, mod_avg)
            decrease_stock(item_list[j], mod_avg, i)
            need_ordering(item_list[j])
```

year range from 1 to 6. These are primarily used in weather_input and will be discussed there. For now, know that it acts as a switch statement. Once weather_input executes and we get weather data for the year, save as weather_list in weather_input, we then deep copy it to a local variable weather_cpy.

The for loop that ranges over item_list walks the program through every item in the list. For every item, calculations are run for every day of the year, which is where the second for loop that runs over the range of 365, or days in the year. Within the daily loop we have several things occur. First finding the modified average of an item. If the average is 10, then that is first modified by the weather reduction. Next, we check to see if the current day is a holiday. If it is, then we multiply mod_avg by 2. Once that is complete, the counter for holiday_incr is incremented. The use of the modulo is to keep the counter between the lengths of the list for the holidays. To look for a holiday outside of the index would cause problems.

resupply_stock(item_list[j], i) takes in the current item and the current day to help determine if an ordered product has arrived. This is checked first as we need to consider shipping time. If this was after an item was ordered and the shipping time was 2 days, then this check would happen and decrease the time. In doing so we would actually lose a day of shipping time. Also,

emergency orders would be ordered, and they would arrive on the same day.
This checks for shipping time only. If the requirement is met, we then enter the if statement, add the value stored in two_week_grace, reset shipping time, noting we received the item, and

```
def resupply_stock(self,day):
    if self.shipping_time == 0:
        self.init_val += self.two_week_grace
        self.shipping_time = 2
        self.shipped = False
        order_list.append(order(self.name,day, self.two_week_grace, self.id))
```

then creating on order object and inserting into order_list[].
Once the resupply check is complete, the program then moves into the weekend check, or weekend_check(). The purpose for this check is to determine the day of the week in which the current day is on. There are limitations to this method. This assumes that all New years falls on a Monday (Monday = 1) as we modulo over 8. This more than likely doesn't affect much as it

```
def weekend_check(self, day):
    day_check = day % 8
    if day_check >= 4:
        self.weekend_mod = (1 + ((day_check * 10)/100))
    elif day_check <= 1:
        self.weekend_mod = (1 - ((day_check * .5)/100))
    else:
        self.weekend_mod = 1
```

should not matter too much where a day sits, only the order in which it falls. For the weekend, weekend_mod is the highest values it can be from 1.4 to 1.7. This marks the highest traffic on weekends. Next, the beginning of the week. These are for values of 0 and 1, Monday and Tuesday respectively. Finally, we have the middle of the week values that get a flat modifier of 1, or no deviation from the mod_avg.
The program then calculates the rolling average of the items use. This is done in find_avg(). find_avg() takes in three values, the object, the day, and how much of the item was used on that day. We check to see if we have enough days saved up in the list, if not we then just add it to the

```
def find_avg(self, day, use):
    if day < two_weeks:
        self.rolling_avg.append(use * self.weekend_mod)
    else:
        self.rolling_avg.append(use * self.weekend_mod)
        self.rolling_avg.pop(0)
        avg = 0
        for i in range(len(self.rolling_avg)):
            avg += self.rolling_avg[i]
        self.two_week_grace = math.ceil((avg/len(self.rolling_avg)) * two_weeks)
```

list. Once there are enough days, an average use can be found. We take the ceiling as if you need 4.6 items, you actually need 5.
Now that we have a rolling average, next is decreasing stock with decrease_stock() function. Just like find_avg(), this takes in the object, the items use, and the day. To calculate how much is used, we take the ceiling of item use multiplied by the weekend modification. This is then subtracted from the stock of the item, init_val. The function then checks if there are no items in stock. If not, then an emergency order is made with emergency_order(). If an item needs to be

```
def decrease_stock(self, item_use, day):
    used =  math.ceil(item_use * self.weekend_mod)
    self.init_val -= used
    if self.init_val <= 0 and self.shipped == False:
        emergency_order(self)
    if self.init_val < (self.store_avg * self.shipping_time) and self.shipped == False:
        self.shipped = True
        self.orders.append([day])
```

ordered, then the conditions of not being shipped, to avoid double orders, and if the store average use multiplied by shipping time is more than our current stock, an order is then added to the item's orders[] and the shipped attribute is changed to true.
The last function call in the loop is need_ordering(). A simple little function that decrease shipping time for an item.

```
def need_ordering(self):
    if self.shipped == True:
        self.shipping_time -= 1
```

With that final function, that completes the loop.

- Order_list.sort(key=lambda x:x.day) – this sorts the order list on the days in which the orders are placed. This way every order is now in order based on date, not the appended order they were placed in.
- Apply_order_num() – Here is where an order gets an order number. If the dates of the current and previous are the same, then they are given the same order number. If they are different, the current item is then incremented, and the comparisons continue.

```
def apply_order_num():
    global num_order
    num_order += 1
    for i in range(len(order_list)):
        if i == 0:
            order_list[i].order_num = num_order
        else:
            if order_list[i].day == order_list[i-1].day:
                order_list[i].order_num = num_order
            else:
                num_order += 1
                order_list[i].order_num = num_order
```

- folder = timeDate() – Here the program gets the current time for naming the files. The is a local as we needed to be able to have a consistent naming scheme to find and alter files. Such as when we go from .xls to .csv.

```
def timeDate():
    now = datetime.now()
    return now.strftime("%H-%M-%S-")
```

- file = makeFileName(folder) – This creates the structure for how items will be saved within the folder that was just created.

```
def makeFileName(folder):
    return os.getcwd()+"\\"+folder+"\items-ordered-"+ timeDate()
```

- xlsToCsv(save_xls(order_list, '2016',file,folder)) – This is a function call within a function call. xlsToCsv() first executes but cannot complete until save_xls() completes. In save_xls(),

```
def save_xls(self,year,file,folder):
    global xlsPos
    file_check(folder)
    workbook = Workbook()
    sheet = workbook.add_sheet('output',cell_overwrite_ok=False)
    sheet.write(0, 0, "Item")
    sheet.write(0, 1, "Item ID")
    sheet.write(0, 2, "Day")
    sheet.write(0, 3, "Quantity")
    sheet.write(0, 4, "Order Number")
    for i in range(len(self)):
        sheet.write(i+1, 0, self[i].item_name)
        sheet.write(i+1, 1, self[i].item_num)
        sheet.write(i+1, 2, intToDate(self[i].day,year))
        sheet.write(i+1, 3, self[i].qty)
        sheet.write(i+1, 4, self[i].order_num)
    xlsPos = len(self)
    #issue with the pi here. maybe
    workbook.save(file+"-"+year+".xls")
    return file+"-"+year
```

order_list[], year (a string value), file, and folder are all sent in. The program checks for the folders existence, if it does not exist it creates it first using file_check(). Next, an instance of

```
#if folder does not exist, then it creates it
def file_check(folder):
    check_folder = os.path.isdir(folder)
    if not check_folder:
        os.makedirs(folder)
```

workbook is created to save the .xls file. Once the contents of order_list[] are inserted into excel, the location is noted and returned to xlsToCsv() so it knows where to look to make the conversion happen.

```
def xlsToCsv(fileName):
    read_file = pd.read_excel(fileName+".xls")
    read_file.to_csv(fileName+".csv",index=None,header=True)
    return fileName
```

- order_list = [] – This is where the list is reinitialized to nothing. This is done as a new year is going to be produced.
- to_splitter = mergXls(folder) – This is where all of the .xls files, from the various years, are merged into one .xls file. It is then converted to .csv using the same method as above.

```
def mergeXls(folder):
    path = os.getcwd() +"\\" + folder
    xls_list = []
    for files in glob.glob(path+"\*.xls"):
        xls_list.append(pd.read_excel(files))
    merged = pd.concat(xls_list,ignore_index=True)
    mergedFile = folder +"\\"+"data_merged"
    merged.to_excel(mergedFile+".xls",index = False)
    xlsToCsv(mergedFile)
    return mergedFile
```

MergedFile is then returned as the location for the merged .csv file. We need this as the next step will be to separate the .csv file into individual items.

- csvsplitter.seperate(to_splitter) - To accomplish this, mergedFile is sent out of data_gen_mk2 and into another program csvsplitter.py.
- os.chdir("..\ML") – This changes the current working directory in Windows to the ML directory.
- os.system("python MultiItemML.py") – Runs the machine learning code.