Assignment 2 companion

2022-2023

On the Manual

This manual will help you through the process of writing your code in Python. Its purpose is twofold. First, it provides you with guidelines on the structuring of your code, and how you can split it up into several manageable parts. Second, it highlights some requirements that we have on your program. These ensure that you create a useful and flexible tool, and simultaneously facilitate checking on our side.

The manual contains the following:

Contents

Preparing your files

All of your code will be contained in a single .py file. For this, we start by making a template file. Please, name your file “s123456-Tool.py”, with your own student number.

This file will contain several elements to help you get started. First, we have selected some useful packages for you. If you plan to use any other packages, please discuss this with us.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math
```

Second, you will find two variables:

```
student_nr = "s123456" # Add your student number, same as in data file names
data_path  =  "..."  #  Example:  'C:/Documents/AssetManagement/Assignment/'  -
Replace with folder path containing your files.
```

These will be used to import your data. Please add your student number to the “student_nr” variable, and add the path to the folder containing your data files to the “data_path” variable. Note that the path should end in a “/”. Finally, the file contains the following function:

```
def run_analysis():

    return

run_analysis()
```

As you see, this function is currently empty. Your tool will consist of multiple functions, which we expect you to call from here. You will get strict guidelines on the expected contents of this function.

Age-based maintenance

Data preparation

In this section, we discuss how you can import the historical failure data that you were given, and how you can prepare this for more complex analysis that you do later. Often in data analytics, this process takes the majority of the time, particularly if you aim to create a tool that works under general circumstances. Your raw data may contain many errors which need to be detected and resolved. However, for this assignment you may assume that your data is generally correct, expediting this process. This section will therefore focus on importing your data, and transforming it to a format suitable for your analysis.

Start by importing the datasets used for the age-based maintenance portion of the assignment. You are given three datasets with historic failure data, which have names like "s123456-Machine-1.csv". We can use pandas to import these datasets. Add the following lines of code to your run_analysis() function.

```
machine_name = 1
machine_data = pd.read_csv(f'{data_path}{student_nr}-Machine-{machine_name}.csv')
```

The first line creates a simple variable which determines the particular machine that we view, which is machine 1 in this case. The second line of code uses the data_path, student_nr and machine_name variables to import the dataset as a pandas dataframe. You will use these dataframes a lot. Therefore, it may be worthwhile to refamiliarize yourself with some of the functions of pandas, most importantly loc and iloc.

If you observe any of your datasets, you should find it has two columns, "Times" and "Event". The first column contains the times at which an event occurred, and the second column contains the type of event that occurred, being "PM" for preventive maintenance, or "failure" for a failure.

data_preparation()

Next, create a function named data_preparation() outside your run_analysis() function. This new function will take your machine_data variable as input, and should alter it in the following way:

1. **Add a "Duration" column to the dataset.** Your dataset contains the times corresponding to each event, but we need the *durations.* These are defined as the times between events.
2. **Sort the dataset on the "Duration" column.** When we calculate the Kaplan-Meier estimator for the reliability function, you want the events to be sorted in the length of the durations, starting with the shortest durations and ending with the longest.
3. **Ensure that duplicate durations are ordered properly.** It is possible, but not guaranteed, that your dataset contains a certain duration more than once. Your code should deal with this, by placing these events in the proper order, based on what you learned in the lectures. Note that even if your dataset does not contain duplicates, your code is still expected to deal with duplicates correctly. It may therefore be helpful to temporarily create a dataset for testing, where you explicitly include such a duplicate duration.

After completing these steps, you may need to reset the indices of your pandas dataframe. Now, the function data_preparation() can return your prepared data, containing the "Times", "Event" and "Durations" columns. The output should look approximately like this (though the values may be different)

```
        Time    Event   Duration
0     916.25       PM       0.22
1     916.71       PM       0.46
2    1360.58       PM       0.51
3    1738.50       PM       2.28
4    1386.93  failure       2.57
```

Finally, to use your function, place the following line of code in the run_analysis() function:

```
prepared_data = data_preparation(machine_data)
```

Kaplan-Meier estimator

In this section, we discuss how you can estimate your Kaplan-Meier reliability function. Programming this will closely follow the method outlined in the slides. To get started, create a function named create_kaplanmeier_data(), which takes your prepared data as an input. In the run_analysis() function, place the following call to this function

```
KM_data = create_kaplanmeier _data(prepared_data)
```

The create_kaplanmeier_data() will add another two columns to your dataframe. The first of these columns will contain the probability of an event at any given time, and the second column contains the reliability function. You can calculate these values by approximately following these steps:

1. **Add a column named "Probability" to your dataframe.** Start by giving each value in this column equal weight, making sure they sum to 1.
2. **Update these probabilities based on the observed events.** One way to achieve this, is to walk through the rows in your dataframe one by one. If the event in a particular row is censored, you need to evenly divide the probability in this row to the remaining rows below it. Finally, you set the probability of the current row to 0.
3. **Merge duplicate durations.** To finalize your probabilities, you need to check if there are any duplicate, uncensored, durations in your dataset. If this is the case, then you need to merge them. To do this, you sum the probabilities of all these duplicates in one row, and set the probabilities of the other duplicates to 0. Of course, it is helpful to remember that any duplicate durations should be next to another in your dataframe, as they were previously sorted.
4. **Calculate the reliability function for each duration.** Add another column to your dataframe, named "Reliability". Since you have your probabilities ready now, calculating the reliability should be a straightforward process. Remember that your reliability function should start at 1, and should decrease with each uncensored event.

You should now have your Kaplan-Meier estimator ready. Make sure your function returns your dataframe, which should now consist of 5 columns. The returned dataframe should look like this:

```
        Time    Event  Duration  Probability   Reliability
0     916.25       PM      0.22     0.000000  1.000000e+00
1     916.71       PM      0.46     0.000000  1.000000e+00
2    1360.58       PM      0.51     0.000000  1.000000e+00
3    1738.50       PM      2.28     0.000000  1.000000e+00
4    1386.93  failure      2.57     0.009524      0.990476
```

With your Kaplan-Meier estimator ready, you are now ready to provide some managerial insights. In the next section we will briefly discuss the MTBF, one of these insights. However, you might find it helpful to instead start with the visualization of the reliability function, which is discussed later in this manual. Working on some figures relating to your reliability function may help you find any errors.

Mean time between failures (Kaplan-Meier)

We expect you to implement the Kaplan-Meier MTBF in a function named meantimebetweenfailures_KM(), taking the dataframe named KM_data as an input. The output should be a single value, the MTBF. To use this function, add the following two lines to your run_analysis() function:

```
MTBF_KM = meantimebetweenfailure_KM(KM_data)
print('The MTBF-KaplanMeier is: ', MTBF_KM)
```

With your Kaplan-Meier data ready, calculating the Kaplan-Meier MTBF should be fairly simple. Remember that the MTBF is calculated as the sum of the multiplication of the durations and the probabilities in your dataframe.

Weibull distribution fitting

***Warning:** This is most likely the hardest part of the assignment. In this section you will determine the best fitting numbers for the lambda and kappa parameters. You will use these numbers again in other functions. If you are stuck, you can still work on the functions after this, by choosing some values for lambda and kappa yourself. Of course, they will not be correct, but trying different values may help you when working on other parts of the code in the meantime.*

In this section, you will fit a Weibull distribution to the dataset. That is, you will find the best lambda and kappa, the two parameters of the Weibull distribution. We will optimize the values of lambda and kappa through **brute-forcing**. This is a method for optimization, in which you essentially try all possible values in a given range. The quality of a solution is then determined through something called **likelihood**, which is a measure to determine how "likely" the data that you have is under a set of parameters. The set of parameters resulting in the highest likelihood is the set of parameters that explains your data the best.

While brute-force methods can be very useful, they have several serious shortcomings. One of these shortcomings is that we cannot possibly evaluate every solution. Since lambda and kappa may be any value above 0, there are infinite possibilities for these parameters. Naturally, you cannot possibly check all these options. Therefore, a brute-force method can only test a limited range of values, which you need to determine beforehand. Second, since brute-force methods rely on trying a lot of different solutions, they may be very computationally expensive. The method typically scales poorly if you have multiple parameters to optimize; even for only 2 parameters (lambda and kappa), you will find that you will need to try many different options.

However, in many cases you may find that brute-force methods are a straightforward method to get very good results.

We continue with the code. First, create a function named fit_weibull_distribution(), which takes your dataframe of the prepared data as an input. Add the following line of code to the run_analysis() function:

```
lamb_val, kap_val = fit_weibull_distribution(prepared_data)
```

Inside the fit_weibull_distribution() function, you should do the following:

1. **Create a variable with the search ranges for lambda and kappa.** To get you started, the value for lambda is between 1 and 35, and we expect you to determine this to the nearest integer (so you will need to check 35 different values). For kappa, the value is between 0.1 and 3.5, and you will need to determine it to the nearest decimal (again, 35 different options). Here we immediately observe a shortcoming of the brute-force method. With 35x35 options, you will need to check a total of 1225 different options. This will require quite some computational power, and may take some minutes depending on your computer. Therefore, until you are ready to get your final results, you may wish to be less precise here, and try less possible values for lambda and kappa.

   Programming these ranges can be done through a function such as numpy.linspace.

2. **Create a dataframe which will contain your likelihood data.** Your dataframe should start with two columns, one for lambda and another for kappa. Each row should correspond to a pair of values for lambda and kappa (meaning you will have 1225 rows in your final calculations).
3. **Create log-likelihood data for each event in your dataset.** This is the complicated part of the function. The goal is to calculate the log-likelihood of any duration for any value of lambda and kappa.

To get started, you will need a place to store this data. We suggest you start by looping over all durations in your dataset. Then, for each duration, you add another column to the dataframe you constructed in step 2.

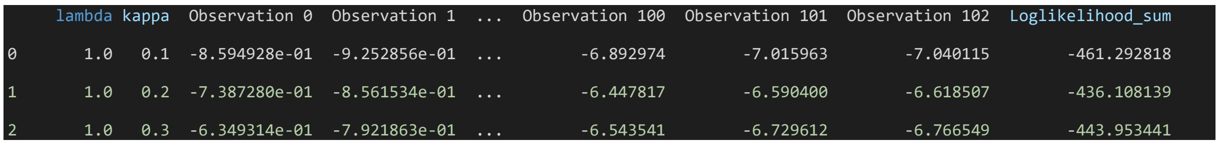
When you add this column, you can immediately start filling its values. To do this, construct another two loops, one looping over all values of lambda, another for kappa. (Remember that each row corresponds to a particular pair of values for lambda and kappa). You should now be three loops deep. Since you are looping over approximately 100 durations, and 35x35 different parameter values, this will require a lot of iterations.

Now you can calculate the log-likelihood for this duration, lambda, kappa trio. This is done either through the log of the Weibull density function, log(f(t)), or through the log of the Weibull reliability function, log(R(t)). The value of t in these functions is the duration. Which of these two options you should use depends on whether the current duration corresponds to a censored event or not. You may find more details on this in the slides, as well as the exact definitions of these functions.

Warning: Your program may give a warning that you are dividing by zero when calculating this log. If everything is programmed correctly, this is not a problem. This happens because the functions f(t) and R(t) may result in 0, and log(0) is not defined. In this case, Python will fill your table with '-inf'. You can resume your calculations as normal, but the likelihood of this set of parameters will be – infinity. As a result, we know that this is not the optimal set of parameters.

4. **Calculate the sum of the log-likelihoods.**
   You should now have a column of log-likelihoods for every duration. Add a final column to your dataframe for the sum of the log-likelihoods. Any element in this column should contain the sum of all the likelihoods of that row. Be sure to not include the column containing lambda and kappa in these sums.

The resulting dataframe should look something like this:



```
   lambda kappa  Observation 0  Observation 1  ...  Observation 100  Observation 101  Observation 102  Loglikelihood_sum
0     1.0   0.1 -8.594928e-01  -9.252856e-01  ...        -6.892974        -7.015963        -7.040115        -461.292818
1     1.0   0.2 -7.387280e-01  -8.561534e-01  ...        -6.447817        -6.590400        -6.618507        -436.108139
2     1.0   0.3 -6.349314e-01  -7.921863e-01  ...        -6.543541        -6.729612        -6.766549        -443.953441
```

Finally, the best values for lambda and kappa are the ones with the highest sum of likelihood values. Find the highest element in the final column of your dataframe, and determine the corresponding lambda and kappa. As a hint, you may use a function like .idxmax() for this. Make sure that your function returns the values of lambda and kappa.

Weibull MTBF and data

Of course, knowing the best values for lambda and kappa is not too useful in itself. Instead, we want to get managerial insights based on these estimates. We separate this into two parts.

MTBF:

First, we want to create a function that calculates the MTBF based on the Weibull distribution. Create a function named meantimebetweenfailures_weibull(). This function takes your calculated values of lambda and kappa as inputs. Add the following line of code to your run_analysis() function:

```
MTBF_weibull = meantimebetweenfailure_weibull(lamb_val, kap_val)
print('The MTBF-Weibull is: ', MTBF_weibull)
```

Calculating the MTBF based on the Weibull distribution should be straightforward. Note that you will need to calculate the gamma function, which can be done through a function in the math library.

Weibull data:

Of course, before we can create a figure containing the Weibull reliability function, we need to calculate the values of R(t) for a range of values for t. Start by creating a function named create_weibull_curve_data(). This function takes your dataset and your calculated values of lambda and kappa as an input. Add the following line of code to your run_analysis() function:

```
weibull_data = create_weibull_curve_data(prepared_data, lamb_val, kap_val)
```

In this function, you will need to do the following:

1. **Create a dataframe with two columns.** The first column should be named "t", and will contain a range of durations. The second column should be named "R_t", and will contain the corresponding values of the reliability function.
2. **Define a range of durations.** We suggest you calculate the reliability function for all values between 0 and the largest duration in your dataset. Of course, you cannot calculate the function R(t) for every value in this range, as there are infinitely many. Thus, you will need to select durations for an arbitrary number of points, say 0, 0.01, 0.02, … etc. Selecting too few will create a poor plot, whereas selecting too many will make your tool slower.
3. **Calculate the reliability function.** Next, calculate the reliability function R_t for each duration. See the slides for the mathematical definition of this function.

We expect the function create_weibull_curve_data() to return your dataframe, which should look approximimately like:

```
           t       R_t
0       0.00       1.0
1       0.01       1.0
2       0.02       1.0
3       0.03       1.0
4       0.04       1.0
...      ...       ...
3888   38.88  0.025257
3889   38.89  0.025192
3890   38.90  0.025128
3891   38.91  0.025064
3892   38.92     0.025
```

Visualization of reliability functions

Before we can evaluate time-based maintenance policies, we should first create figures containing both our reliability functions. Create a function named visualization(), taking the Kaplan-Meier data and Weibull data as inputs. To run this function, you can add the following line of code to your run_analysis() function.

```
visualization(KM_data, weibull_data)
```

Note that the visualization() function does not need to return a value.

In this function, you should make a single figure, showing both the Kaplan-Meier estimate of the reliability function, and the Weibull reliability function. For this, we suggest you use matplotlib, which you should be familiar with. You may find the .step() function in matplotlib useful for plotting the Kaplan-Meier reliability function.

Of course, you need to ensure that your plots are of high-quality and contain labels, a title, etc. They should look good for any dataset that your tool analyses.

Finally, save the figure in the folder containing your dataset. Your file should be named “s123456-Machine-1-Reliability.png”, of course with your student number and the current machine number. You can use the following line of code to get the correct filename and location.

```
file_location = f'{data_path}{student_nr}-Machine-{machine_name}-Reliability.png'
```

Age-based maintenance

With all the preparation out of the way, we can finally start the analysis of a maintenance policy. For this, we will create a single function which will achieve the following:

1. The function calculates the cost rate for a range of maintenance ages.
2. The function plots these cost rates.
3. The function returns the optimal maintenance age and the corresponding cost rate.

Create a function named create_cost_data(). This function will take many inputs, being your prepared_data variable, your Weibull estimates for lambda and kappa, the preventive maintenance cost for this machine, and the corrective maintenance cost for this machine. Add the following lines to your run_analysis() function:

```
PM_cost = 5
CM_cost = 20
best_age, best_cost_rate = create_cost_data(prepared_data, lamb_val, kap_val, PM_cost, CM_cost)
print('The optimal maintenance age is ', best_age)
print('The best cost rate is ', best_cost_rate)
```

Note that the values of PM_cost and CM_cost are placeholders, we will extract these values from a dataset later. For now, you can place the correct costs here manually, or leave them as is.

create_cost_data()

The logic in your create_cost_data() should approximately be as follows:

1. **Define a range of maintenance ages.** Similar to when you created durations for the Weibull data, you need to select a range of maintenance ages to check and plot.
2. **Calculate the values of F(t) and R(t) for each maintenance age.** The mean cycle cost and mean cycle length of your policy are calculated through the Weibull distribution function F(t) and the Weibull reliability function R(t), so you will need to calculate these. It may be helpful to create a pandas dataframe to collect this data.
3. **Calculate the mean cost per cycle for each maintenance age.** If you have F(t) and R(t) ready, this should be straightforward.
4. **Calculate the mean cycle length.** As you may remember from the slides, the mean cycle length is equal to the integral of R(t) from 0 to t. This integral does not necessarily have an explicit solution, so you will need to approximate it. We do this through a **Riemann sum**. Details on the Riemann sum are in the slides.
   It may be helpful to remember that you already calculated R(t) for many different values in step 2. So, if in step 1 you selected a range of maintenance ages with a precision of 0.01, you may want to select your width Δ as 0.01 too.
5. **Calculate the cost rate for each maintenance age.** This value is equal to the mean cost per cycle divided by the mean cycle length.
6. **Create a plot of the cost rates**. We want showing how the maintenance age impacts the cost rate. Of course, this should be a high-quality plot. A very important note here, is that for very small maintenance ages, the cost rate will be extremely high. So, if you include these values, your plot will do a poor job of showing the cost rates at relevant values. You should come up with a way to ensure that your plot only shows relevant values for any dataset. Save this file as "s123456-Machine-1-Costs.png", with your student number and the correct machine name.

7. **Determine the optimal maintenance age and the corresponding cost rate.** Make sure your function returns both of these values.

Assuming you have completed all steps before this, your tool should be ready to analyse age-based maintenance policies. To recap, your run_analysis() function should now look like:

```python
def run_analysis():
    #Data preparation
    machine_name = 1
    machine_data = pd.read_csv(f'{data_path}{student_nr}-Machine-{machine_name}.csv')
    prepared_data = data_preparation(machine_data)

    #Kaplan-Meier estimation
    KM_data = create_kaplanmeier_curve_data(prepared_data)
    MTBF_KM = meantimebetweenfailure_KM(KM_data)
    print('The MTBF-KaplanMeier is: ', MTBF_KM)

    #Weibull fitting
    lamb_val, kap_val = fit_weibull_distribution(prepared_data)
    weibull_data = create_weibull_curve_data(prepared_data, lamb_val, kap_val)
    MTBF_weibull = meantimebetweenfailure_weibull(lamb_val, kap_val)
    print('The MTBF-Weibull is: ', MTBF_weibull)

    #Visualization
    visualization(KM_data, weibull_data)

    #Age-based maintenance optimization
    PM_cost = 5
    CM_cost = 20
    best_age, best_cost_rate = create_cost_data(prepared_data, lamb_val, kap_val, PM_cost, CM_cost)
    print('The optimal maintenance age is ', best_age)
    print('The best cost rate is ', best_cost_rate)
    return
```

We will come back to this later for some minor improvements, most notably to make sure the user inputs the values of “machine_name”, “PM_cost” and “CM_cost”.

Condition-based maintenance

Now that your tool can evaluate age-based maintenance policies for a range of datasets, we want to implement a condition-based maintenance policy. Of course, this requires a different dataset, and will again need some preparation. You will find that this is very similar to the previous data preparation section.

Data preparation

First, we want to import historical condition data from your .csv file. This file should be named

“s123456-Machine-1-condition-data.csv”, of course with the correct student and machine number. Add the following lines of code to your run_analysis() function:

```
condition_data = pd.read_csv(f'{data_path}{student_nr}-Machine-{machine_name}-condition-data.csv')
prepared_condition_data = CBM_data_preparation(condition_data)
```

You will also need to make a function named CBM_data_preparation().

CBM_data_preparation():

Your condition information dataset should contain two columns. The first is a column named “Times”, which shows the time at which a condition was measure. Times are measured at regular intervals. The second column is named “Condition”, which is a numeric value indicating the state of the machine at that time.

You will optimize a condition-based maintenance policy through simulation. To do this, we need to know how the condition-information of the machine increases over time. So, you need to add a column to your dataframe, named “Increments”. This column should contain the increases in the “Condition” column.

You may have already seen that the “Condition” column is not always increasing. When the machine is repaired, the condition is reset to 0. As a result, your increment will be negative. You need to filter any rows with negative increments from your dataframe. After this, you may need to reset the indices for your pandas dataframe.

When you have completed these steps, you can return the dataframe. It should look like this:

```
     index   Time  Condition  Increments
0        0   0.00       0.00        0.00
1        1   0.10      11.12       11.12
2        2   0.20      12.98        1.86
3        3   0.30      17.37        4.39
4        4   0.40      19.41        2.04
```

This should complete the data preparation for the CBM policy.

Failure level

Before we can start with the analysis of a CBM policy, you need to determine one extremely important value. There is a fixed condition-level at which your machine breaks down. You should be able to extract this value from the dataset. We expect you to create a variable named failure_level in the run_analysis() function, which contains this value. You may create a new function to calculate this, though it is also possible to extract this value with one line of code.

Creating deterioration paths

This section contains the most complicated part of the analysis. You will create a function that will simulate the deteriorating machine. Doing so, you can create many different deterioration paths, which you can use to evaluate a maintenance policy. If you provide a maintenance threshold, a value above which you perform preventive maintenance, you can then see how many simulations ended in a failure, and how many ended in a preventive maintenance action.

Create a function named CBM_create_simulations (). This function takes three inputs. The first is your pandas dataframe, containing all the historical condition data and the increments you calculated. The second is the failure level. The third, is some maintenance threshold. For now, an educated guess will suffice for this. Add the following code to the run_analysis() function:

```
threshold = 0.8*failure_level
simulation_data = CBM_create_simulations(CBM_prepared_data, failure_level, threshold)
```

Here, we take a threshold value equal to 80% of the failure level as an initial guess.

CBM_create_simulations ()

We continue with our CBM_create_simulations() function. This function should do the following:

1. **Define a number of simulations.** The more simulations you run, the smoother your resulting graphs will be. However, this will come at a significant computational cost. To make sure you are not waiting too long, you can start by doing about 100 simulations. However, in later sections, you may find that this results in a poor graph, and you will need to increase the number of simulations.
2. **Create a dataframe for your simulations.** Your dataframe should contain two columns, "Duration" and "Event". The first column will contain the time between any two events, and the second will contain the type of event that occurred.
3. **Simulate deterioration paths.** Now, you will need to create your deterioration paths. First, create a loop which iterates once for each simulation that you run.

   Creating a deterioration path will require a few things. First, each path starts at a condition of 0, and at a time of 0. Then, you increase your time by some fixed value. When this time increases, your machine condition will increase by a random value. To get a random value, you can sample an increment from your dataset. For this, you may consider the numpy.random.choice() function.

   Next, you check if your machine has failed. If this is the case, the current time is the duration, and the event is a failure. You can then add this to your dataframe from step 2. If your machine has not failed, you can check if you do preventive maintenance instead. This will require you to compare the current condition of the machine to your threshold. If you do preventive maintenance, you again know the duration of this simulation, and you know that your event was preventive maintenance.

   If no failure or preventive maintenance took place, you increase the time again, and you sample another condition increment. Of course, you do not know how many increments may take place before an event occurs. To implement this logic, you will need to create an infinite loop. Each iteration of this loop increases the time, increments the condition, and checks if an event takes place. You end the loop when an event takes place. To do this, you can use "while True:" to create an infinite loop, and you can use the "break" keyword to stop your loop.

At the end of your loop, your dataframe should contain an additional duration and event. Then, you set the condition information and time to 0 again, and you start the next simulation. This continues until all your simulations are done. Make sure to return your dataframe as output, which should look similar to this.

```
    Durations   Event
0         8.2      PM
1         6.0      PM
2         7.1      PM
3        10.8 failure
```

CBM policy evaluation

Now that you have a set of samples for a single threshold, you need to analyse the resulting cost rate. Create a function named CBM_analyse_costs(). This function takes three inputs. First, it takes your simulation data. Second, it takes the preventive maintenance cost. And third, it takes the corrective maintenance cost. Add the following code to run_analysis()

```
cost_rate = CBM_analyse_costs(sample_data, PM_cost, CM_cost)
print('The cost rate is ', cost_rate)
```

Inside your CBM_analyse_costs() function, you need to implement the following logic:

1. **Calculate the average cycle duration.** This should be based on your simulated durations.
2. **Calculate the fraction of cycles ending in failure.** Again, base this information on your simulated durations.
3. **Calculate the average cost per cycle.** You can find how to do this in the slides on CBM.
4. **Calculate the cost rate.** As with an age-based policy, this is equal to the average cost per cycle divided by the average cycle length.

Your function should return the cost rate.

CBM policy optimization:

To finalize our analysis of the condition-based maintenance policy, we need to determine the optimal condition threshold. To do this, we take a similar approach to determining the optimal maintenance age for age-based maintenance policies.

Create a function named CBM_create_cost_data(). This function takes four inputs, the first being your **prepared condition data.** Remember that we are NOT referring to your simulations, but to the output of the data preparation section. The second and third inputs of your function are the preventive and corrective maintenance costs, and the final input is the failure level.

Also, you can remove some parts of the run_analysis() function. In particular, you can remove the lines you added in the previous section, being:

```
threshold = 0.8*failure_level
simulation_data = CBM_create_simulations(CBM_prepared_data, failure_level, threshold)

cost_rate = CBM_analyse_costs(sample_data, PM_cost, CM_cost)
print('The cost rate is ', cost_rate)
```

Instead, we will call these functions from within CBM_create_cost_data(). You can replace this part of code with the following:

```
CBM_cost_rate, CBM_threshold = CBM_create_cost_data(prepared_condition_data, PM_cost, CM_cost, failure_level)
print('The optimal cost rate under CBM is ', CBM_cost_rate)
print('The optimal CBM threshold is ', CBM_threshold)
```

We continue with the contents of the CBM_create_cost_data() function. This function will take care of the following logic:

1. **Set a range of thresholds that you will evaluate.** Remember that you will run many simulations for each of these thresholds, so you need to limit how many thresholds you evaluate. We expect you to find the optimal maintenance threshold to the nearest integer value.
2. **Create a set of simulations for each threshold**. You can call the function CBM_create_simulations() for each threshold, as you have already prepared this function before.
3. **Evaluate the cost rate of each threshold.** Now that you have the simulation data for each threshold, you can use your CBM_analyse_costs() function to evaluate the costs for each threshold. You may want to do this in the same loop as step 2.
4. **Plot the cost rates.** You need to create a high-quality figure of the cost rates for different maintenance thresholds. You will find that creating such a plot will provide similar problems as creating the figure with cost rates for the time-based maintenance policy. In particular, very small maintenance thresholds will result in enormous costs, and includes this data will result in a poor figure. Your figure should look good for any dataset. If you find your figure is not very smooth, then you will need to increase the number of simulations that you do in CBM_create_simulations(). Save this file as “s123456-Machine-3-CBM-Costs.png”, with your student number and the correct machine number.
5. **Determine the optimal maintenance threshold and the corresponding cost rate**. Since you calculated the cost rate for each threshold, this is as simple as finding the threshold with the lowest cost rate. Make sure to return both the threshold and the cost rate as output of your function.

This completes the analysis of the CBM policy. What remains is some general programming to make sure that your tool handles data correctly.

Input handling

The main functionality of your tool should now be complete. However, if someone want to use your tool to analyse a specific machine, they still need to change variables within your code. Of course, we don't want this. So, we should ask the user some questions about their analysis instead. In total, you should ask four questions to the user:

1. Which machine do they want to analyse? In your test data, this can be a 1, 2 or 3, corresponding to the machine numbers.
2. What is the preventive maintenance cost for this machine? Responses should be a number.
3. What is the corrective maintenance cost for this machine? Responses should again be a number.
4. Is there condition information for this machine/do we want to optimize a CBM policy as well? The answer should be "yes" or "no", without capitalization.

Ask these questions from within your run_analysis() function. You can read the responses from the user with the input() function.

The responses to these questions should of course be handled. That is, you should make sure that the variables "machine_name", "PM_cost" and "CM_cost" take the correct values based on the answers. Additionally, when the answer to question 4 is "yes", you should make sure that the functions used for age-based AND condition-based maintenance are called. If the answer is "no", your analysis should only evaluate the age-based maintenance policy.