# CS4215 Quantitative Performance Evaluation for Computing systems
# Assignment 2

Marco, Schouten
m.schouten-4@student.tudelft.nl
Student Number: 5352908

October 17th, 2021

## 1 Exercise (10 Points), *Jackson*

1. In a Jackson Network with N nodes, we can analyze each node independently with the condition:

$$\rho_i = \frac{\lambda_i}{k_i \mu_i} < 1, \forall i = 1..N$$

$$\mu = \text{service rate}, \lambda = \text{arrival rate}, k = \text{number of servers},$$

Then we compute the arrival rates for the first node.

$$\lambda_1 = r_1 + 1/3\lambda_2$$

$$\lambda_2 = r_2 + 1/3\lambda_1 + 1/3\lambda_2$$

Then we have that (in the schematic $r_2 = 1$):

$$\lambda_2 = 3/2 + 1/2\lambda_1$$

Substituting:

$$\lambda_1 = r_1 + 1/3(3/2 + 1/2\lambda_1)$$

Then:

$$\lambda_1 = (6/5)r_1 + 3/5$$

Now we can evaluate the condition

$$\rho_1 = \frac{\lambda_1}{k_1 \mu_1} < 1$$

$$\frac{(6/5)r_1 + 3/5}{3} < 1$$

$$\mathbf{r_1 < 2}$$

2. For a Jackson Network we define response time of a job is defined as the time from when the job arrives to the network until it leaves the network, including possibly visiting the same server or different servers multiple times Given the following requirement on $r_1$ $r_1 = 0.9r_1^{max} = 9/5$ We can compute $lambda_i$

$$\lambda_1 = 54/25 + 3/5 = 69/25 = 2.76$$
$$\lambda_2 = 3/2 + 1/2(69/25) = 2.88$$

We see that:

$$\rho_1 = \frac{2.76}{3} = 0.92$$
$$\rho_2 = \frac{2.88}{5} = 0.576$$

For a Jackson Network:

$$\mathbb{E}[N_1] = \frac{\rho_1}{1-\rho_1} = \frac{0.92}{1-0.92} = 11.5$$
$$\mathbb{E}[N_2] = \frac{\rho_2}{1-\rho_2} = \frac{0.576}{1-0.576} = 1.36$$
$$\mathbb{E}[N_{tot}] = \mathbb{E}[N_1] + \mathbb{E}[N_2] = 11.5 + 1.36 = 12.86$$

Note that throughput $X = \lambda_1 + \lambda_2 = 2.76 + 2.88 = 5.64$
Therefore by Little's Law we conclude that:

$$\mathbb{E}[T] = \frac{\mathbb{E}[N]}{X} = \frac{12.86}{5.64} = 2.28$$

$$\mathbb{E}[\mathbf{T}] = \mathbf{2.28}$$

Alternative I compute each piece independently :

$$\mathbb{E}[T_1] = \frac{\mathbb{E}[N_1]}{\lambda_1} = \frac{11.5}{2.76} = 4.17$$
$$\mathbb{E}[T_2] = \frac{\mathbb{E}[N_2]}{\lambda_2} = \frac{1.36}{2.88} = 0.47$$

I compute weighted Average:

$$\mathbb{E}[T] = \frac{\lambda_1\mathbb{E}[T_1] + \lambda_2 * (\mathbb{E}[T_1] + \mathbb{E}[T_2])}{\lambda_1 + \lambda_2} = 2.28$$

# 2 Exercise 2. (15 Points), *simulate G/G/1*

Note that the difference between the simulated and theoretical waiting time for G/G/1 is 1.03.

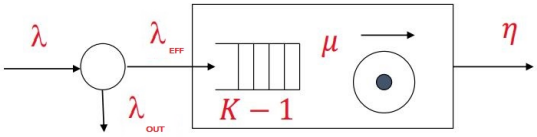
Whereas the difference between G/G/1 and M/M/1 is much about 20 times more.

*python code is in the appendix*

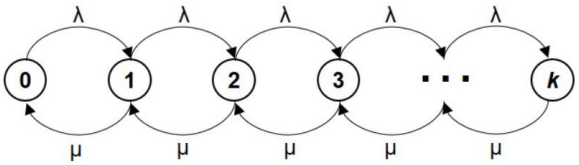
| | **Waiting Time** |
|---|---|
| M/M/1 formulas | 1.921 |
| G/G/1 formulas | 20.713 |
| G/G/1 simulated | 18.223 |

# 3 Exercise 3. (15 Points) *M/M/1/N queue*

Given the following system:




We draw the corresponding CTMC




Note, there always exists stochastic equilibrium, regardless of the relationship between $\lambda$ and $\mu$. This system cannot be unstable by construction

1. First we derive the limiting probabilities $\pi_i$ solving the system:

$$\pi Q = 0$$
$$\sum_i \pi_i = 1$$

Expanding the terms:

$$\pi_1 = \rho\pi_0$$
$$\pi_2 = \rho\pi_1 = \rho^2\pi_0$$
$$...$$
$$\pi_k = \rho\pi_{k-1} = \rho^k\pi_0$$
$$\pi_0(1 + \rho + \rho^2 + ... + \rho^k) = 1$$

Thus we obtain:

$$\pi_0 = [1 + \sum_{n=1}^{k}(\frac{\lambda}{\mu})^n]^{-1} = [1 + \frac{(\frac{\lambda}{\mu})(1 - (\frac{\lambda}{\mu})^k)}{1 - \frac{\lambda}{\mu}}]^{-1}$$

set $\rho = \lambda/\mu$

$$\pi_0 = \frac{1-\rho}{1-\rho^{k+1}}$$
$$\pi_n = \frac{1-\rho}{1-\rho^{k+1}}\rho^n$$

Now we can retrieve the closed form solution for E[N]:

$$E[N] = \sum_{n=0}^{K} n\rho^n \pi_n = \frac{1-\rho}{1-\rho^{k+1}} \sum_{n=0}^{K} n\rho^n$$

Note that:

$$\sum_{n=0}^{K} n\rho^n - \rho(\sum_{n=0}^{K} n\rho^n) = (1+\rho+\rho^2+...+\rho^k) - k\rho^{k+1} = \frac{\rho(1-\rho^k)}{1-\rho} - k\rho^{k+1}$$

Therefore:

$$\sum_{n=0}^{K} n\rho^n = \frac{\rho(1-\rho^k)}{(1-\rho)^2} - k\frac{\rho^{k+1}}{1-\rho}$$

Substitute in the E[N] and you finally get:

$$E[N] = \frac{\rho}{1-\rho^{k+1}}[\frac{1-\rho^k}{1-\rho} - k\rho^k]$$

2. To get a closed form E[T] we have to refer to the effective input that actually gets inside the system, that is determied by:

$$\lambda_{eff} = \lambda Pr N < K$$

Given that:

$$Pr\{N = K\} = \frac{1-\rho}{1-\rho^{k+1}}\rho^k$$

We have:

$$\lambda_{eff} = \lambda[1 - \frac{1-\rho}{1-\rho^{k+1}}\rho^k] = \lambda\frac{1-\rho^k}{1-\rho^{k+1}}$$

From Little's Law we can determine E[T]

$$E[T] = \frac{E[N]}{\lambda_{eff}} = \frac{1/\mu}{1-\rho^k}[\frac{1-\rho^K}{1-\rho} - k\rho^k]$$

3. For M/M/1/2, (i.e. K=2)

$$E[T_{1,0}] = \pi_1 * \text{service time} = \frac{\pi_1}{\mu} = \frac{1-\rho}{1-\rho^{2+1}}\rho$$
$$= \frac{1-\rho}{1-\rho^3}\rho$$

meaning the probability of being in state 1 (i.e. having 1 job in the system) times the time of completing one job.

# 4 Exercise 4. (10 Points), M/M/k

First we compute $P_Q$

$$\begin{aligned}
P_Q &= Pr\{\text{arrival finds all servers busy }\} \\
&= Pr\{\text{arrival sees} \geq \text{jobs in the system}\} \\
&= \text{Limiting probability that there are} \geq \text{k jobs in the system} \\
&= \sum_{i=k}^{\infty} \pi_i \\
&= k^k/k!\pi_0 \sum_{i=k}^{\infty} \rho^i \\
&= \frac{(k\rho)^k \pi_0}{k!(1-\rho)}, \text{where } \pi_0 = [\sum_{i=0}^{k-1} \frac{(k\rho)^i}{i!} + \frac{(k\rho)^k)}{k!(1-\rho)}]^-1
\end{aligned}$$

Then

$$\begin{aligned}
E[N_Q] &= \sum_{i=k}^{\infty} \pi_i (i-k) \\
&= \pi_0 \sum_{i=k}^{\infty} \frac{\rho^i k^k}{k!}(i-k) \\
&= \pi_0 \frac{\rho^k k^k}{k!} \sum_{i=k}^{\infty} \rho^{i-k}(i-k) \\
&= \pi_0 \frac{\rho^k k^k}{k!} \sum_{i=0}^{\infty} \rho^i i \\
&= \pi_0 \frac{\rho^k k^k}{k!} \rho \frac{1}{(1-\rho)2} \\
&= P_Q \frac{\rho}{1-\rho}
\end{aligned}$$

Therefore:

$$E[T_Q] = \frac{E[N_Q]}{\lambda} = P_Q \frac{\rho}{(1-\rho)\lambda}$$

$$E[T] = E[T_Q] + \frac{1}{\mu} = P_Q \frac{\rho}{(1-\rho)\lambda} + \frac{1}{\mu}$$

$$E[N] = \lambda E[T] = P_Q \frac{\rho}{(1-\rho)} + k\rho$$

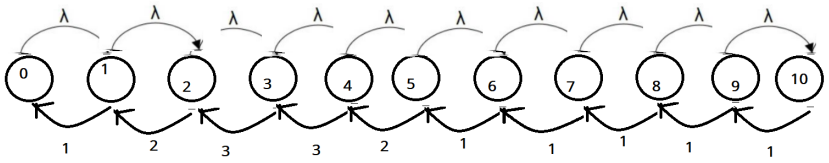
| | K=1 | K=2 | K=4 | K=8 | K=16 | K=32 | K = 64 |
|---|---|---|---|---|---|---|---|
| $\frac{E[N_Q]}{E[N]}$ | 0.5 | 0.2 | 0.05 | 0.01 | 9.9e-04 | 1.8e-05 | 1.33e-08 |
| $E[T_Q]$ | 1.18 | 0.29 | 0.07 | 0.01 | 1.1e-03 | 2.1e-05 | 1.57e-08 |

Table 1: table for (i) the fraction of customers that are delayed and (ii) the expected waiting time for those customers who are delayed.

The trend is clear, the higher the K , the less customers get stuck in queue.

# 5 Exercise 5. (15 Points) *M/M/1/PS*

1. First we draw the CTMC for the system.




Then we compute the Q matrix

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -0.9500 | 0.9500 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1.0000 | -1.9500 | 0.9500 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 2.0000 | -2.9500 | 0.9500 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 3.0000 | -3.9500 | 0.9500 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 3.0000 | -3.9500 | 0.9500 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 2.0000 | -2.9500 | 0.9500 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1.0000 | -1.9500 | 0.9500 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1.0000 | -1.9500 | 0.9500 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.0000 | -1.9500 | 0.9500 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.0000 | -1.9500 | 0.9500 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.0000 | -1.0000 |

We solve the system for $\pi_i$

$$\pi Q = 0$$
$$\sum_i \pi_i = 1$$

Then we obtain

$$\Pi = \{1/11, 1/11, 1/11, 1/11, 1/11, 1/11, 1/11, 1/11, 1/11, 1/11, 1/11\}$$

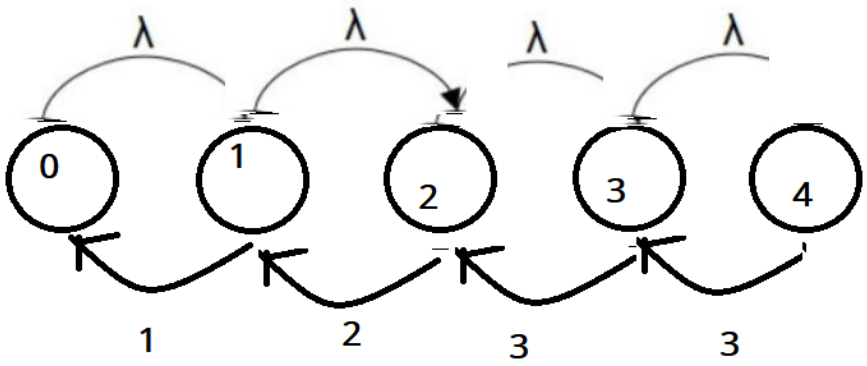
Then we compute

$$E[N] = \sum_i i\pi_i = 55 * \frac{1}{11} = 5$$

Lastly, through Little's Law we obtain:

$$E[T] = \frac{E[N]}{\lambda} = \frac{5}{0.95} = 5.26$$

2. First we draw the CTMC for the system.




Then we compute the Q matrix

| | | | | |
|---|---|---|---|---|
| -0.9500 | 0.9500 | 0 | 0 | 0 |
| 1.0000 | -1.9500 | 0.9500 | 0 | 0 |
| 0 | 2.0000 | -2.9500 | 0.9500 | 0 |
| 0 | 0 | 3.0000 | -3.9500 | 0.9500 |
| 0 | 0 | 0 | 3.0000 | -3.0000 |

We solve the system for $\pi_i$

$$\pi Q = 0$$
$$\sum_i \pi_i = 1$$

Then we obtain

$$\Pi = \{0.2, 0.2, 0.2, 0.2, 0.2, 0.2\}$$

Then we compute

$$E[N] = \sum_i i\pi_i = 10 * 0.2 = 2$$

Now we have to examine the effective $\lambda$ that goes in the system (due to the FCFS queue)

$$\lambda_{eff} = \lambda * Pr\{n < 4\} = 0.95 * 0.8 = 0.76$$

Lastly, through Little's Law we obtain:

$$E[T] = \frac{E[N]}{\lambda_{eff}} = \frac{2}{0.76} = 2.63$$

# 6 Exercise 6. (35 Points)

1. Hyper parameter search has been conducted with a Grid Search of the model hyper parameters. Each configuration has been evaluated using Repeated Stratified KFold Cross Validation (split 3, repetitions 3). Preprocessing pipeline included re-sampling to have an even number of each output class, Sacaled to 0 mean, and unit variance. As Bonus, part I have added a dimensionality reduction algorithm (principal component analysis) to reduce feature space. To get an overall variance of at least 90%, the minimum number of principal components is 16.

   - **Logistic Regression**
     Parameter Search
     space['solver'] = ['newton-cg', 'lbfgs', 'liblinear']
     space['penalty'] = ['none', 'l1', 'l2', 'elasticnet']
     space['C'] = loguniform(1e-5, 100)
     Best Accuracy: 0.789
     Best Hyperparameters: 'C': 0.008301451461243866, 'penalty': 'none', 'solver': 'newton-cg'
   - **Support Vector Machine**
     Parameter Search:
     space['gamma'] = [1e-2, 1e-1, 1, 10, 100]
     space['kernel'] = ['linear', 'poly', 'rbf', 'sigmoid']
     space['C'] = [1e-2, 1e-1, 1, 10, 100]
     Best Accuracy: 0.8961169778667685
     Best Hyperparameters: 'C': 100, 'gamma': 0.01, 'kernel': 'rbf'
   - **Decision Tree**
     Parameter Search: space['criterion'] = ['gini','entropy']
     space['splitter'] = ['best', 'random']
     space['max depth'] = [2,21,32]
     space['min samples split'] = [0.1, 0.2, 0.4, 0.6]
     Best Score: 0.9277611940298506
     Best Hyperparameters: {'criterion': 'entropy', 'max depth': 21, 'min samples split': 0.1, 'splitter': 'best'}

- **Random Forest**
  Parameter Search:
  space['bootstrap'] = [ True, False ]
  space['max depth'] = [2,21,32]
  space['max features'] = [ 8, 12, 16 ]
  space['min samples leaf'] = [0.01, .1, 0.2, ]
  space['min samples split'] = [0.1, 0.4, 0.6]
  space['n estimators'] = [ 10, 50 , 100 ]
  Best Score: 0.931639104858943
  Best Hyperparameters: {'bootstrap': False, 'max depth': 21, 'max features': 16, 'min samples leaf': 0.01,
  'min samples split': 0.1, 'n estimators': 100}
- **MLP**
  Parameter Search:
  space['hidden layer sizes'] = [ (10, 3) , (3,10), (5,5) ]
  space['activation'] = ['relu' , 'tanh' ,'logistic']
  space['solver'] = ['bfgs', 'sgd', 'adam']
  space['learning rate init'] = [0.001, 0.01, 0.0001]
  Best Score: 0.9198506758782127
  Best Hyperparameters: {'activation': 'tanh', 'hidden layer sizes': (10, 3), 'learning rate init': 0.01, 'solver': 'adam'}

2. **Feature importance**

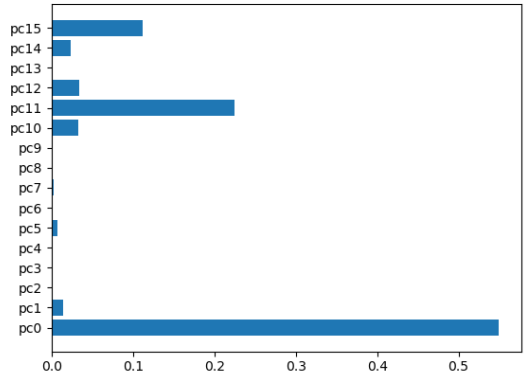
Entropy is, according to Shannon, the uncertainty in the outcome of an experiment. The greater the degree of uncertainty, the greater the value of the Entropy. So for a uniform distribution, i.e. every single output is equiprobable, the entropy value will be greater. Conversely, for a non-uniform (biased) distribution, the entropy value will be lower, because some outputs are more likely than others. For example, the entropy for a fair coin: $p_1 = 0.5$, $p_2 = 0.5$

$$Entropy_1 = \sum_i -p_i log_2 p_i = -0.5 \cdot log_2 \cdot 0.5 - 0.5 \cdot log_2 \cdot 0.5 = 1 \quad (1)$$

Instead for a very rigged coin: $p_1 = 0.01$, $p_2 = 0.99$

$$Entropy_2 = \sum_i -p_i log_2 p_i = -0.01 \cdot log_2 \cdot 0.01 - 0.99 \cdot log_2 \cdot 0.99 = 0.08 \quad (2)$$

By choosing the attribute that maximizes the Information Gain, in the next step, we obtain a lower entropy. This implies that the probability of the various outcomes in the Subset $S_k$ follow a certain tendency. The information gain however numerically quantifies how effective it is to ask about any attribute $a_i$ in order to obtain a more biased distribution of probabilities of the outcomes.




# 7 Appendix

## 7.1 code for 3.2

```
import numpy as np
import pandas as pd



def main():
    # read data
    df= pd.read_csv('memory-usage.csv', sep='\t')
    X = df.iloc[:, 0:3].values
    feat_cols = ['size', 'speed', 'inter_arrival']
    df = pd.DataFrame(X,columns=feat_cols)


    # remove outlier
    print('size  before' + str(df.shape))
    idx_to_remove = np.argmin(df.loc[:, 'speed'])
    df = df.drop([df.index[idx_to_remove]])
    print('size after ' + str(df.shape))



    # get lambda
    arrivals = df.loc[:,'inter_arrival']
    avg_arrival_time = np.average(df.loc[:,'inter_arrival'])
    lambda_ = 1 / avg_arrival_time
```

```
print('lambda: ' +str(lambda_))

# get mu
job_size = df.loc[:, 'size']
speed = df.loc[:, 'speed']
service_time = np.divide(job_size, speed)
avg_service_time = np.average(service_time) # avg_service_time= tau
mu_ = 1 / avg_service_time
print('mu: ' +str(mu_))

# compute rho
rho_ = lambda_ / mu_

# get Ca
std_arrivals =  np.std(df.loc[:,'inter_arrival'])
Ca_ = std_arrivals / avg_arrival_time

# get Cs
std_service_time =  np.std(service_time)
Cs_ =std_service_time / avg_service_time



#__________________ PART 1 - FORMULAS
# waiting time for M/M/1
wait_MM1_formulas = rho_ / (mu_ - lambda_)


# waiting time for G/G/1
rho_= lambda_ / mu_
wait_GG1_formulas = ( rho_ / (1- rho_) )* ((Cs_**2 + Ca_**2)/2) * avg_service_time


#__________________ PART 2 - SIMULATIONS
# arrivals
# service_time

arrivals = np.array(arrivals)
service_time = np.array(service_time)


# init
n = arrivals.size
A = np.zeros(n)
S = np.zeros(n)
C = np.zeros(n)
```

```
    W = np.zeros(n)

    # first iteration is hardcoded
    A[0] = arrivals[0]
    S[0] = A[0]
    C[0] = S[0] + service_time[0]
    W[0] = C[0]  -A[0]

    # repeat
    for i in range(1, n):
        A[i] = A[i-1] + arrivals[i]
        S[i] = np.maximum(C[i-1], A[i])
        C[i] = S[i] + service_time[i]
        W[i] = C[i] - A[i]

    wait_GG1_simulation = np.average(W)

    wait_MM1_formulas = np.round(wait_MM1_formulas, 3)
    wait_GG1_formulas = np.round(wait_GG1_formulas, 3)
    wait_GG1_simulation = np.round(wait_GG1_simulation, 3)

    print(np.round(wait_GG1_simulation - wait_GG1_formulas, 2) )


    print('wait_MM1_formulas: {} \nwait_GG1_formulas: {} \nwait_GG1_simulation: {} '.format(


    print('done')




if __name__ == "__main__":
    main()
```

## 7.2 code for 3.4

```
import numpy as np
# what I want
# rate E[Nq] / E[N]
# E[Tq]

K = 16

MU = 0.85
```

```
RHO = 0.50
LAMBDA = RHO * K * MU



# (0) evaluate Pi_0
sum = 0
for i in range(0, K):
    sum += np.power(K*RHO, i) / np.math.factorial(i) + (np.power(K*RHO, K)) / (np.math.facto
PI_0 = 1 /sum

# (1) evaluate Pq
Pq =  ((np.power(K*RHO, K)) *  PI_0) / (np.math.factorial(K) *(1-RHO))

# (2) evaluate E[Nq] =  Pq * rho / (1-rho)
Nq =   Pq * RHO / (1-RHO)

# (3) evaluate E[Tq] =  E[Nq] / LAMBDA
Tq = Nq /LAMBDA

# (4) evaluate E[T] =  E[Tq] + 1/MU
T = Tq + (1/MU)

# (5) evaluate E[N] =  E[T] * LAMBDA
N = T * LAMBDA

# (6) RATIO = rate E[Nq] / E[N]
RATIO = Nq / N


print('Ratio: {} \n Tq: {}'.format(np.round(RATIO,10), np.round(Tq,10) ))
```

### 7.3 code for 3.6

```
import time


from scipy.stats import loguniform
from sklearn.model_selection import RepeatedStratifiedKFold, GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
import pandas as pd  # used to load the data
import numpy as np  # optimized numerical library
from scipy import stats
from sklearn import preprocessing, metrics, utils, decomposition, model_selection, linear_mo
# library providing several ML algorithms and related utility
```

```
from sklearn.neural_network import MLPClassifier
from imblearn import over_sampling  # provides several resampling techniques to cope with u
from imblearn.over_sampling import RandomOverSampler
from collections import Counter
import matplotlib.pyplot as plt


# Start by defining three helper functions:
# - one to plot the sample distribution  acorss the class labels (to see how un-/balanced t
# - one to compute and plot the confusion matrix
# - one to plot data in 2D with different colors per class label

def plot_pie(y, labels, title=""):
    target_stats = Counter(y)
    sizes = list(target_stats.values())
    explode = tuple([0.1] * len(target_stats))

    fig, ax = plt.subplots()
    ax.set_title(title + " (size: %d)" % len(y))
    ax.pie(sizes, explode=explode, labels=target_stats.keys(), shadow=True, autopct='%1.1f%%
    ax.axis('equal')


def compute_and_plot_cm(ytest, ypred, labels, title=""):
    global nfigure
    # Compute confusion matrix
    cm = metrics.confusion_matrix(ytest, ypred)

    accuracy = metrics.accuracy_score(ytest, ypred, normalize=True)

    # Normalize the matrix
    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    print(cm)

    # Plot the confusion matrix

    nfigure = nfigure + 1
    plt.figure(nfigure)  # new numbered figure
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)  # plot the confusionmatrix
    plt.title("Confusion Matrix Normalized (%s) Accuracy: %.1f%%" % (title, accuracy * 100))
    plt.colorbar()  # plot the color bar as legend

    # Plot the x and y ticks using the class label names
    tick_marks = np.arange(len(labels))
    plt.xticks(tick_marks, labels, rotation=45)
```

```
    plt.yticks(tick_marks, labels)


def plot_2d(xpred, ypred, labels, title=""):
    global nfigure
    # define the colors to use for each class label
    colors = ['red', 'blue', 'green', 'yellow', 'black']
    len_colors = len(colors)
    if len_colors < len(labels):
        print("WARNING: we have less colors than classes: some classes will reuse the same c

    nfigure = nfigure + 1
    plt.figure(nfigure)  # new numbered figure
    plt.title("Feature Space (%s)" % title)  # add title

    # plot each class label with a separate color
    for c in [4, 5]:
        cur_class = (ypred == c)  # get all points belonging to class c
        plt.plot(xpred[cur_class, 0], xpred[cur_class, 1], 'o', color=colors[c % len_colors]


nfigure = 0  # used to number the figures


 #_____________________________________________ (1) LOAD DATA _____________________________
# Get the dataset loaded and define class labels
data = pd.read_csv('jobs.csv', header=0)
data_class_labels = ["successful", "unsuccessful"]
# print(data.head())

# All data columns except last are input features (X), last column is output label (y)
n_features = len(data.columns) - 1

X = data.iloc[:,0:n_features].values
y = data.iloc[:,n_features].values
X = X[y != 2]
y = y[y != 2]
X = X[y != 3]
y = y[y != 3]

# plot_pie(y, data_class_labels, "Original")
# plt.show()

# _____________________________________________ (2) BALANCE DATA ___________________________
################ Resample data #################
# Google data is very skewed, try to balance the dataset
```

```
# sm = over_sampling.SMOTE(random_state=42, ratio="auto")
# X_balanced, y_balanced = sm.fit_sample(X,y)
# ros = RandomOverSampler(random_state=42, sampling_strategy='auto')
# X_res, y_res = ros.fit_resample(X, y)
#
# # Plot the balanced label distribution
# plot_pie(y,data_class_labels, "Balanced")
# plt.show()
ros = RandomOverSampler(random_state=42, sampling_strategy=0.99)
X_res, y_res = ros.fit_resample(X, y)
# plot_pie(y_res,data_class_labels, "Balanced")
# plt.show()

# ____________________________________________ (3) SMALLER SUBSET ________________________
# Resample the data with simple random resampling (if too big)
# - replace decideds if sampling with or without replacement
# - n_samples decide the size of the ouput: if set to None ouput = input (i.e. no resampling
X_small, y_small = utils.resample(X_res,y_res, replace=False, n_samples=5000)
# Plot the resampled label distribution
# plot_pie(y_small,data_class_labels, "Sampled")
# plt.show()




# ____________________________________________  remove outliers ___________________________
# data_SMALL_dict = {'c1':  X_small[:,0],
#         'c2':  X_small[:,1],
#         'c3':  X_small[:,2],
#         'c4': X_small[:, 3],
#         'c5': X_small[:, 4],
#         'c6': X_small[:, 5],
#         'c7': X_small[:, 6],
#         'c8': X_small[:, 7],
#         'c9': X_small[:, 8],
#         'c10': X_small[:, 9],
#         'c11': X_small[:, 10],
#         'c12': X_small[:, 11],
#         'c13': X_small[:, 12],
#         'c14': X_small[:, 13],
#         'c15': X_small[:, 14],
#         'c16': X_small[:, 15],
#         'c17': X_small[:, 16],
#         'c18': X_small[:, 17],
```

```
#         'c19': X_small[:, 18],
#         'c20': X_small[:, 19],
#         'c21': X_small[:, 20],
#         'c22': X_small[:, 21],
#         'c23': X_small[:, 22],
#         'c24': X_small[:, 23],
#         'c25': X_small[:, 24],
#         'c26': X_small[:, 25],
#         'c27': X_small[:, 26],
#         'c28': X_small[:, 27],
#         }
# df_small = pd.DataFrame(data_SMALL_dict)
# df_small['y'] = y_small
# print(df_small.head())
# print(df_small.shape)
#
# z_scores = stats.zscore(df_small)
# abs_z_scores = np.abs(z_scores)
# filtered_entries = (abs_z_scores < 3).all(axis=1)
# new_df = df_small[filtered_entries]
# # print(new_df.head())
#
#
# X_o = data.iloc[:,0:n_features].values
# y_o = data.iloc[:,n_features].values
# plot_pie(y_o,data_class_labels, "outliers")
# plt.show()



# ____________________________________________ (4)SCALE ____________________________________
################ Scale data ####################
# Train a scaler to standardize the features (zero mean and unit variance)
X_small = preprocessing.StandardScaler().fit_transform(X_small)
# print(np.mean(X_small),np.std(X_small))



# ____________________________________________ (4) DIMENSIONALITY  REDUCTION ___________
# Train a PCA with k dimensions: Write a script to find minimum number of k components that
# contain 90% of variance description

feat_cols = ['feature'+str(i) for i in range(X_small.shape[1])]
small_df = pd.DataFrame(X_small,columns=feat_cols)
# print(small_df.head())
```

```
n_components=16
pca_small = decomposition.PCA(n_components)
principalComponents_small = pca_small.fit_transform(X_small)
pca_cols = ['pc'+str(i) for i in range(n_components)]
principal_small_Df = pd.DataFrame(data = principalComponents_small, columns = pca_cols)
# print(principal_small_Df.tail())
# print('Explained variation per principal component: {}'.format(pca_small.explained_varian
sum_Var = np.sum(pca_small.explained_variance_ratio_) * 100
print('SumVar'+str(np.round(sum_Var,0)))
X_small = principal_small_Df.iloc[:,0:n_components]


# ____________________________________________ (4) SPLIT DATA ____________________________
# Split data in training and testing for 0.33 ratio of testing
X_train, X_test, y_train, y_test = model_selection.train_test_split(X_small, y_small, test_s
print('X train {}'.format(X_train.shape) )
print('X test{}'.format(X_test.shape))
print('y train {}'.format(y_train.shape))
print('y train {}'.format(y_test.shape))


# ____________________________________________ (5) MODEL HYPERPARAMETER SEARCH __________
# LOGISTIC REGRESSION
model =  linear_model.LogisticRegression()
# define evaluation
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define search space
space = dict()
space['solver'] = ['newton-cg', 'lbfgs', 'liblinear']
space['penalty'] = ['none', 'l1', 'l2', 'elasticnet']
space['C'] = loguniform(1e-5, 100)
# define search
search = RandomizedSearchCV(model, space, n_iter=500, scoring='accuracy', n_jobs=-1, cv=cv,
# execute search
result = search.fit(X_small, y_small)
# summarize result
print('Best Score: %s' % result.best_score_)
print('Best Hyperparameters: %s' % result.best_params_)



# SUPPORT VECTOR MACHINE
model = svm.SVC()
cv = RepeatedStratifiedKFold(n_splits=3, n_repeats=1, random_state=1)
space = dict()
space['gamma'] = [1e-2, 1e-1, 1, 10, 100]
```

```
space['kernel'] = ['linear', 'poly', 'rbf', 'sigmoid']
space['C'] = [1e-2, 1e-1, 1, 10, 100]
search = GridSearchCV(model, space, scoring='accuracy', n_jobs=-1, cv=cv)
result = search.fit(X_train, y_train)
print('Best Score: %s' % result.best_score_)
print('Best Hyperparameters: %s' % result.best_params_)



# DECISION TREE
model = tree.DecisionTreeClassifier()
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=1)
space = dict()
space['criterion'] = ['gini','entropy']
space['splitter'] = ['best', 'random']
space['max_depth'] = [2,21,32]
space['min_samples_split'] = [0.1, 0.2, 0.4, 0.6]
search = GridSearchCV(model, space, scoring='accuracy', n_jobs=-1, cv=cv)
result = search.fit(X_train, y_train)
print('Best Score: %s' % result.best_score_)
print('Best Hyperparameters: %s' % result.best_params_)



# RANDOM FOREST
model =  ensemble.RandomForestClassifier()
cv = RepeatedStratifiedKFold(n_splits=3, n_repeats=3, random_state=1)
space = dict()
space['bootstrap'] = [ True, False ]
space['max_depth'] = [2,21,32]
space['max_features'] = [ 8, 12, 16 ]
space['min_samples_leaf'] = [0.01, .1, 0.2, ]
space['min_samples_split'] = [0.1, 0.4, 0.6]
space['n_estimators'] = [ 10, 50 , 100 ]

search = GridSearchCV(model, space, scoring='accuracy', n_jobs=-1, cv=cv)
result = search.fit(X_train, y_train)
print('Best Score: %s' % result.best_score_)
print('Best Hyperparameters: %s' % result.best_params_)


# feature_importance
rf =  ensemble.RandomForestClassifier( n_estimators= 100, bootstrap= False, max_depth= 21, m
                                       min_samples_leaf= 0.01, min_samples_split =0.1 )
rf.fit(X_train, y_train)
plt.barh(pca_cols, rf.feature_importances_)
```

```
print(rf.feature_importances_)
plt.show()



#  MLP CLASSIFIER
model = MLPClassifier()
cv = RepeatedStratifiedKFold(n_splits=3, n_repeats=1, random_state=0)
space = dict()
space['hidden_layer_sizes'] = [ (10, 3)  , (3,10),  (5,5) ]
space['activation'] = ['relu' , 'tanh' ,'logistic']
space['solver'] = ['bfgs', 'sgd', 'adam']
space['learning_rate_init'] = [0.001, 0.01, 0.0001]
search = GridSearchCV(model, space, scoring='accuracy', n_jobs=-1, cv=cv)
result = search.fit(X_train, y_train)
print('Best Score: %s' % result.best_score_)
print('Best Hyperparameters: %s' % result.best_params_)
```