Group Coursework Submission Form

Specialist Masters Programme

Please list all names of group members: (Surname, first name) 1. Nesvik, Even 2. Windsand, Gunnar 3. Viermyr, Mikkel	4. Soh, Avery 5. Amirgaliyeva, Dina **GROUP NUMBER:** **8**
MSc in: Business Analytics	
Module Code: SMM641	
Module Title: Revenue Management and Pricing	
Lecturer: Oben Ceryan	**Submission Date:** 29/03/19

Declaration:

Marker's Comments (if not being marked on-line):

Deduction for Late Submission:

Final Mark: %

Revenue and Pricing Coursework 2 Grp 8

March 29, 2019

Revenue and Pricing Coursework 2
Group 8
Date 29/03/19

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import norm
import matplotlib.patches as mpatches
from matplotlib.patches import Rectangle
```

In [2]:
```
# Data frame stuff
data = []
data = pd.DataFrame(data)
data["choice"] = (0,1,2)
data["meaning"] = ("no purchase","high fare","low fare")
data["prob"] = (0.1,0.3,0.6)
data["price"] = (0,200,100)
data

# Creating the simulations data which we will be using for the rest of the coursework
sel = [np.random.choice(3, 200, p=[0.1, 0.3, 0.6]) for i in range (1000)]
```

Ps! Results might differ because a seed was not set and this causes some randomness in the data which alter the figures etc.

1 Creating base functions for use in a) and b)

In [3]:
```
# Individual revenue calculator

def idv_rev (x):
    ref = []
    data_low= []
    data_high = []
    data_nope = []
    for p in range (len(sel)):
        peta = sel[p]
```

```
            low = 0
            high = 0
            nope = 0
            protect = x
            capacity = 100 - protect

            for i in range(len(sel[0])):

                # take the high fare demand from the protection capacity first
                if peta[i] == 1:

                    if protect > 0:
                        high += 1
                        protect -=1

                    elif capacity > 0:
                        high += 1
                        capacity -=1
                    else:
                        continue

                # If protection is filled or if it is not a high fare demand,
                # Fill the regular capacity based on high or low fare demand
                elif peta[i] == 2:

                    if capacity > 0:
                        low += 1
                        capacity -= 1
                    else:
                        continue
                else:
                    nope += 1
            # Revenue calculations
            total_rev = 200*high + 100*low
            data_low.append(low)
            data_high.append(high)
            data_nope.append(nope)

            # Record  average revenue
            ref.append(total_rev)

        return ref
```

In [4]:
```
# Average Revenue Calculator

def avg_rev (x):
    ref = []
    data_low= []
```

```
        data_high = []
        data_nope = []
        for p in range (len(sel)):
            peta = sel[p]
            low = 0
            high = 0
            nope = 0
            protect = x
            capacity = 100 - protect

            for i in range(len(sel[0])):

                # take the high fare demand from the protection capacity first
                if peta[i] == 1:

                    if protect > 0:
                        high += 1
                        protect -=1

                    elif capacity > 0:
                        high += 1
                        capacity -=1
                    else:
                        continue

                # If protection is filled or if it is not a high fare demand,
                # Fill the regular capacity based on high or low fare demand
                elif peta[i] == 2:

                    if capacity > 0:
                        low += 1
                        capacity -= 1
                    else:
                        continue
                else:
                    nope += 1
            # Revenue calculations
            total_rev = 200*high + 100*low
            data_low.append(low)
            data_high.append(high)
            data_nope.append(nope)

            # Record  average revenue
            ref.append(total_rev)
            avg_rev = np.average(ref)
        return avg_rev
```

```
In [5]: # Low Fare Dataset
```

```python
def low_dataset (x):
    ref = []
    data_low= []
    data_high = []
    data_nope = []
    for p in range (len(sel)):
        peta = sel[p]
        low = 0
        high = 0
        nope = 0

        for i in range(len(sel[0])):

            if peta[i] == 1:
                high += 1
            elif peta[i] == 2:
                low += 1
            else:
                pass
        # Revenue calculations
        total_rev = 200*high + 100*low
        data_low.append(low)
        data_high.append(high)
        data_nope.append(nope)

    return data_low
```

In [6]:

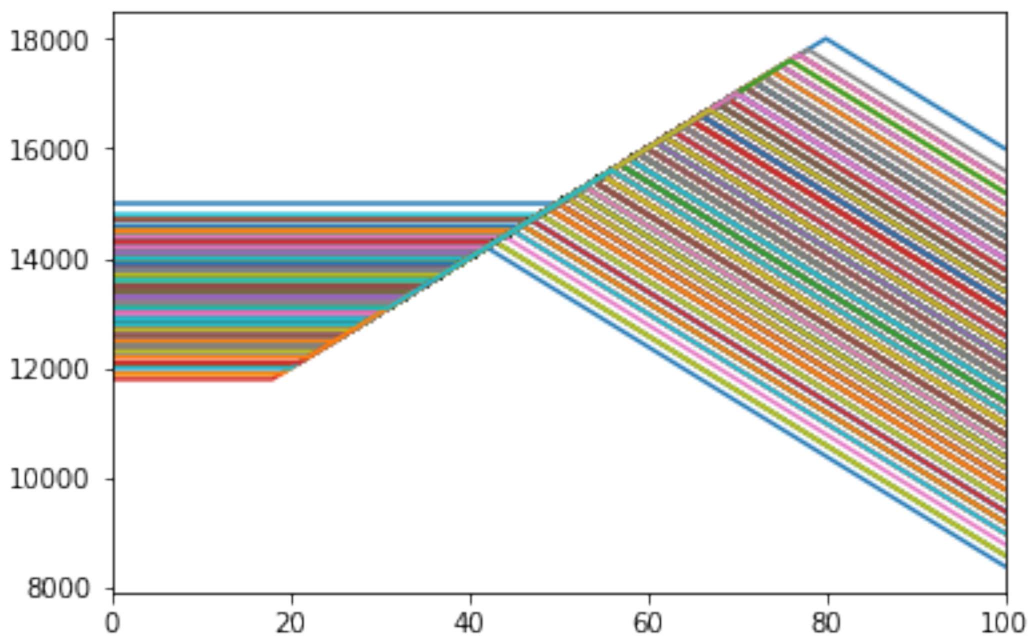
```python
# high Fare Dataset (no capacity criteria)

def high_dataset (x):
    ref = []
    data_low= []
    data_high = []
    data_nope = []
    for p in range (len(sel)):
        peta = sel[p]
        low = 0
        high = 0
        nope = 0

        for i in range(len(sel[0])):

            if peta[i] == 1:
                high += 1
            elif peta[i] == 2:
                low += 1
            else:
```

```
            pass
        # Revenue calculations
        total_rev = 200*high + 100*low
        data_low.append(low)
        data_high.append(high)
        data_nope.append(nope)

    return data_high
```

2 Part A

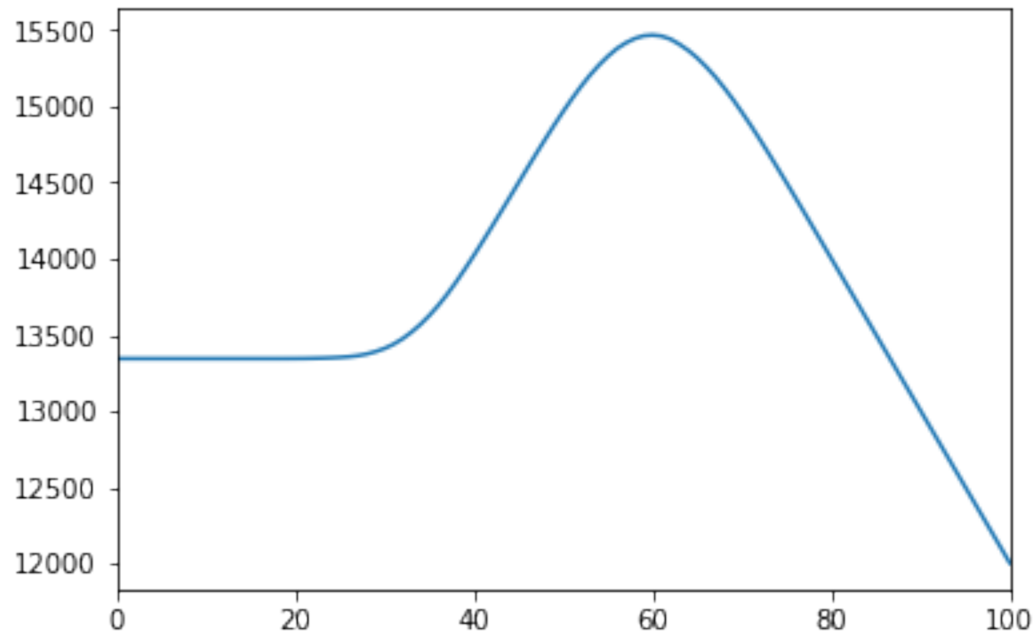
In [7]:
```
# Plot all simulations

pd.DataFrame([idv_rev(i) for i in range(101)]).plot(legend=False)
```

Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x19206f6b00>



In [8]:
```
# Plotting for part A
part_a_rev = pd.DataFrame([avg_rev(i) for i in range(101)])
part_a_rev.plot(legend = False)
```

Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x1a2166e9e8>




```
In [9]: part_a_rev.max()

Out[9]: 0    15468.0
        dtype: float64

In [10]: # Value of revenue at x = 60
         part_a_rev.loc[60]

Out[10]: 0    15468.0
         Name: 60, dtype: float64
```

Part A plot analysis

The optimal protection level here we can see is x = 60 achieving the highest revenue. This is aligned with the probability of getting a high fare at probability of 0.3. Within the plot at x < 30 shows a constant region since the protective level is not yet activated, once it is, reservation for high fare customers have started. Past x > 60 we can see that the average revenue begins to fall, this is due to the protection level for high fare is greater than the available demand of high fare seats. we begin to see seats going unsold due to over protection and hence a fall in revenue. The minimum revenue we should achieve at this rate is 60*200 = 12 000.

3 Part B

3.1 Low Fare Class Distribution

```
In [11]: # df_low = pd.DataFrame([low_dataset(i) for i in range(101)])
         # Unsure part here if there is need to implement the protection level or just take th
```

```
# since the whole point of little wood is to find the optimal booking limit
df_low = pd.DataFrame(low_dataset(0))

# Get the distribution for Low Fare Class
df_low.describe()
```

Out[11]:

	0
count	1000.000000
mean	119.818000
std	7.170655
min	98.000000
25%	115.000000
50%	120.000000
75%	125.000000
max	143.000000

3.2 High Fare Class Distribution

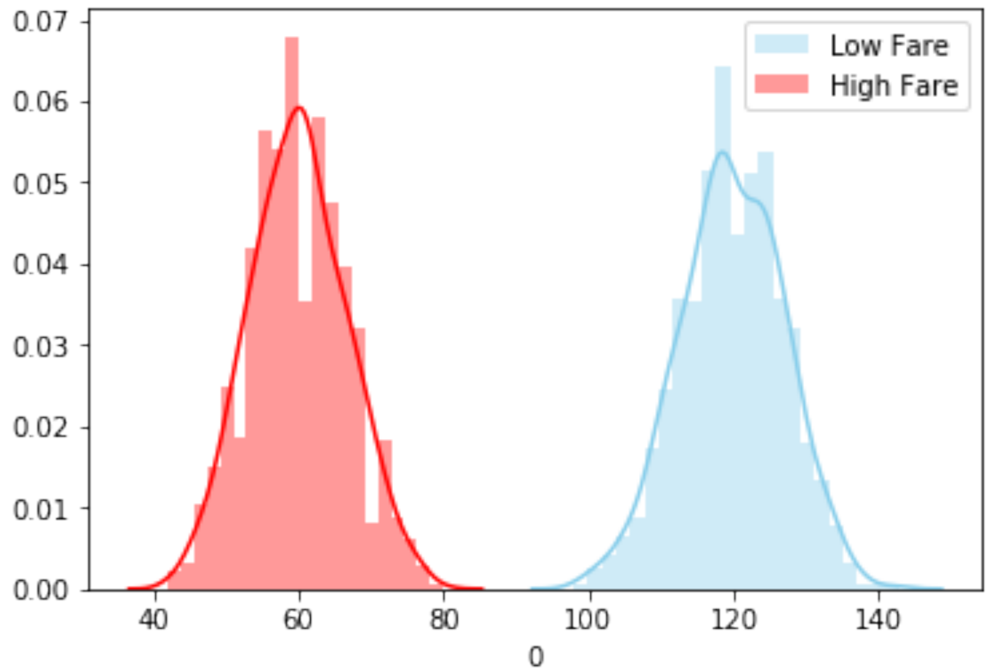
In [12]:
```
# df_high = pd.DataFrame([high_dataset(i) for i in range(101)])
df_high = pd.DataFrame(high_dataset(0))

# Get the distribution for high fare class
df_high.describe()
```

Out[12]:

	0
count	1000.000000
mean	60.013000
std	6.666879
min	42.000000
25%	55.750000
50%	60.000000
75%	65.000000
max	80.000000

3.3 Plot the distributions

In [13]:
```
# Import library and dataset
import seaborn as sns
df = sns.load_dataset('iris')

sns.distplot( df_low[0] , color="skyblue", label="Low Fare")
sns.distplot( df_high[0] , color="red", label="High Fare")
plt.legend()

plt.show()
```



Fare class distributions
Both fare classes share very similar distributions however, we can see that

1. The low fare class has a higher mean and hence the distribution is pushed right
2. The two fare classes do not intersect which is true since we work under the assumption that at any point in time t =200, at most one class can be slected.

3.4 Littlewood part

In [14]:
```
low_rev = 100 # low fare revenue
low_mean = df_low.mean() # mean for low fare
low_sdev = df_low.std() # standard deviation for low fare

high_rev = 200 # high fare revenue
high_mean = df_high.mean() # mean for high fare
high_sdev = df_high.std() # standard deviation for high fare

capacity = 100


# LittleWoods Rule

# 1-low_rev/high_rev
a = 1-(low_rev/high_rev)
Booking_limit = capacity - norm.ppf(a,high_mean,high_sdev)
Booking_limit
```

```
Protection_level = 100-Booking_limit
Protection_level
```

Out[14]: array([60.013])

LittleWood Interpretation

We can see that the optimal protection level produced by littlewood method is 60. This is aligned with the protection level optimisation we achieved in part a where we get the highest revenue around x = 60

4 Part C

In [15]: data

Out[15]:

	choice	meaning	prob	price
0	0	no purchase	0.1	0
1	1	high fare	0.3	200
2	2	low fare	0.6	100

In [16]:

```
###########################################
#        Optimal Dynamic Programming       #
###########################################

high_p = data["price"][1]
low_p = data["price"][2]
t = 200
c = 100
v = np.zeros((c+1,t+1))
outcome = np.zeros((c+1, t+1))

for i in range(1, c+1):
    for j in range(1, t+1):

        # Reject and accept conditions
        reject = v[i,j-1]
        accept = v[i-1,j-1]

        # Probabilities from each of the options
        prob_nope = data["prob"][0]
        prob_high = data["prob"][1]
        prob_low = data["prob"][2]

        # Creating the utility function we get from accepting a high fare vs a low fa
        # By this logic, and on assumption that we always accept high fares
        # We will only accept a low fare if the utility to accept low fare > utility
        utility_h = high_p + accept
        utility_l = low_p + accept
```

```
        # Returns given the following probability
        # Suppose class 0 is selected, we take prob*reject since no capacity is fille
        r_nope = prob_nope * reject

        # We assume that as long as we have capacity, we will accept high paying cust
        r_high = prob_high * (utility_h)

        # Using this utility function, we decide whether or not to accept or reject a
        r_low = prob_low * max(utility_l, reject)

        # Populating the decision outcome array
        if utility_l > reject:
            outcome[i,j] = 1
        else:
            outcome[i,j] = 0

        # Populate v with the returns
        v[i,j] = r_nope+r_high+r_low


v.max()
```

Out[16]: 15980.404074515787

In [17]:
```
pd.DataFrame(outcome).head()
```

Out[17]:

```
     0    1    2    3    4    5    6    7    8    9  ...  191  192  193  194  \
0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0  0.0  0.0
1  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0  0.0  0.0
2  0.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0  0.0  0.0
3  0.0  1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0  0.0  0.0
4  0.0  1.0  1.0  1.0  1.0  1.0  1.0  0.0  0.0  0.0  ...  0.0  0.0  0.0  0.0

   195  196  197  198  199  200
0  0.0  0.0  0.0  0.0  0.0  0.0
1  0.0  0.0  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  0.0  0.0  0.0
3  0.0  0.0  0.0  0.0  0.0  0.0
4  0.0  0.0  0.0  0.0  0.0  0.0

[5 rows x 201 columns]
```

In [18]:
```
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(1, 1, 1)
ax.set_title("Optimal acceptance/rejection decision for the low fare class")
ax.set_ylabel("Seats Remaining at the start of the period(c)")
ax.set_xlabel("Time Remaining(t)")
col = "RdYlGn"
p = ax.matshow(outcome, cmap = col)
```

```
p

class AnyObject(object):
    pass

class AnyObjectHandler(object):
    def legend_artist(self, legend, orig_handle, fontsize, handlebox):
        x0, y0 = handlebox.xdescent, handlebox.ydescent
        width, height = handlebox.width, handlebox.height
        patch = mpatches.Rectangle([x0, y0], width, height, facecolor='red',
                                   edgecolor='black', lw=1,
                                   transform=handlebox.get_transform())
        handlebox.add_artist(patch)
        return patch

class AnyObject1(object):
    pass

class AnyObjectHandler1(object):
    def legend_artist(self, legend, orig_handle, fontsize, handlebox):
        x0, y0 = handlebox.xdescent, handlebox.ydescent
        width, height = handlebox.width, handlebox.height
        patch = mpatches.Rectangle([x0, y0], width, height, facecolor='Green',
                                   edgecolor='black', lw=1,
                                   transform=handlebox.get_transform())
        handlebox.add_artist(patch)
        return patch

plt.legend([AnyObject(),AnyObject1()], ['Low Fare Rejected', "Low Fare Accepted"],
           handler_map={AnyObject: AnyObjectHandler(),AnyObject1:AnyObjectHandler1()})
```

```
Out[18]: <matplotlib.legend.Legend at 0x1a21b7e828>
```

Optimal Dynamic Programming Comments
We can see that the plot follows the logical inclination.

- *Time logic*. If there is less time remaining, we will have an increased sense of urgency and accept low fare customers as represented by the green (accept) region nearer as time remaining approaches 0.
- *Capacity logic*. If there are less seats remaining, we will prioritise those seats for high fare paying customers hence reject the low fare customers. However, this only triggers past a certain time period.

All in all we have a method driven by the theory of urgency driven by time and capacity prioritising high fare, thus we begin to see this lagged diagonal relationship. Additionally, when we have no seats left or no time left, we need to reject the low fare customer. Hence at that point it is a red reject region.

5 Part D

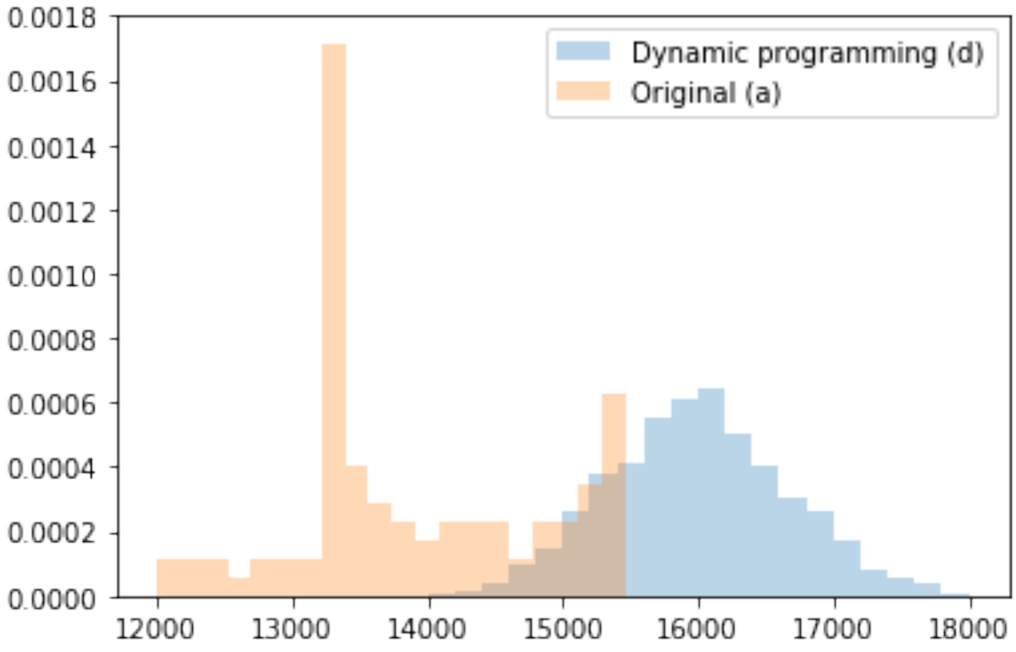
In [19]:
```
dyna_rev =[]

data_outcome = pd.DataFrame(outcome)

# Re running in the simulations
for i in range (0,1000):
    choice = sel[i]
    capacity = 100
    high_fare = 0
    low_fare = 0
    # Implementing a secondary criteria based on results from the dynamic programming
```

```
    for j in range(1,200):
        if capacity > 0:
            if choice[j] == 1:
                high_fare +=1
                capacity -=1

            # Accept low fare only if it is found in the output dataset
            elif choice[j] == 2:
                if outcome[capacity][200-j] == 1:
                    low_fare +=1
                    capacity -=1

        rev = 200*high_fare + 100*low_fare

    dyna_rev.append(rev)

ave_rev = np.average(dyna_rev)
```

In [20]:
```
ave_rev
```

Out[20]: 15951.9

In [21]:
```
plt.hist(dyna_rev, label ="Dynamic programming (d)",histtype = "stepfilled", alpha = (
plt.hist(part_a_rev[0], label = "Original (a)",histtype = "stepfilled", alpha = 0.3, (

plt.legend(loc='upper right')
plt.show()
```



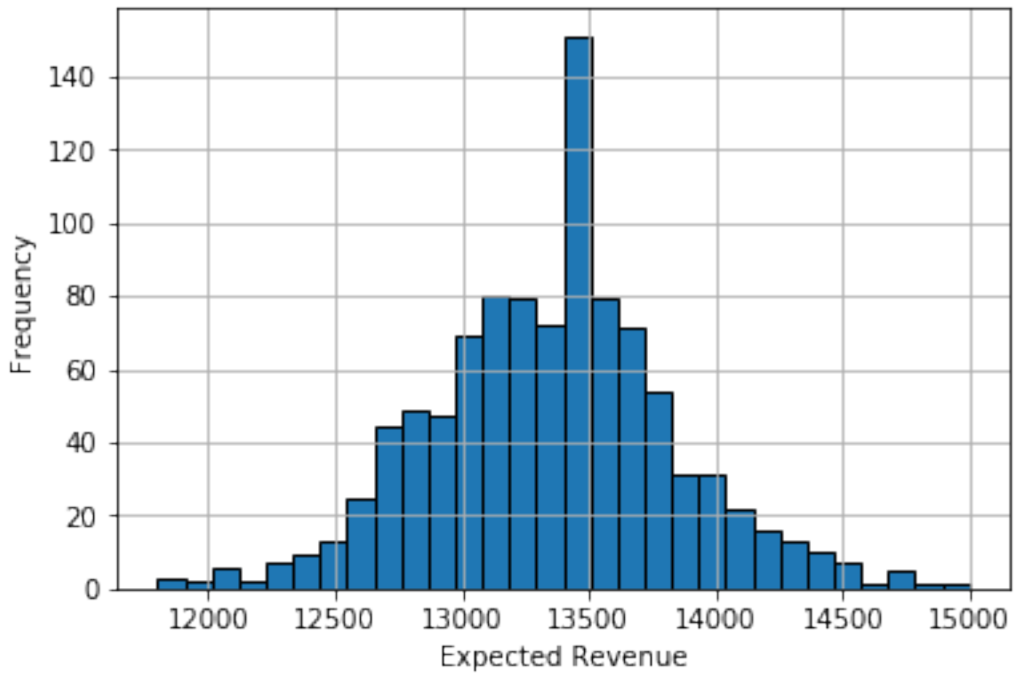
Above we visualise the differences between using dynamic programming and our original model in a) in a plot showing the expected revenue (y-axis) vs the protection level (x-axis)

6 Part E

In [22]:
```
# First come first serve revenue calculator
first_serve = []
data_low= []
data_high = []
data_nope = []

for p in range (len(sel)):
    peta = sel[p]
    low = 0
    high = 0
    nope = 0
    protect = 0
    capacity = 100 - protect

    for i in range(len(sel[0])):

        if protect > 0:

            if peta[i] ==1:
                high += 1
                protect -= 1
            else:
                continue
        elif capacity > 0:

            if peta[i] == 1:
                high += 1
                capacity -=1
            elif peta[i] == 2:
                low += 1
                capacity -=1
            else:
                continue
        else:
            break
        # Revenue calculations
    total_rev = 200*high + 100*low
    data_low.append(low)
    data_high.append(high)
    data_nope.append(nope)

    # Record revenue
```

```
first_serve.append(total_rev)
```

```
In [23]: plt.hist(first_serve, bins = 30, edgecolor ="black")
         plt.xlabel("Expected Revenue")
         plt.ylabel("Frequency")
         plt.grid(True)
         plt.show()
```



Above we visualise the result when disregarding the observed protection level.

7 Part F

```
In [24]: ############################################
         #          Custom Criteria Chooser         #
         ############################################

         def consec(x):
             for i in range(0,1000):
                 consec_data = sel[i]
                 rev=[]
                 capacity = 100
                 high_fare = 0
                 low_fare = 0
                 w=0
```

```
        for j in range(0,200):
            if capacity > 0:
                if consec_data[j] == 1:
                    capacity -= 1
                    high_fare += 1
                    w += 1
                elif consec_data[j] == 2:
                    if w < x:
                        low_fare += 1
                        capacity -= 1
                    else: # Here we reject a low fare
                        w -= 1

        revenue = high_fare*200+low_fare*100
        rev.append(revenue)

        ave_rev = np.average(rev)

    return ave_rev
```

Explanation for Custom Criteria

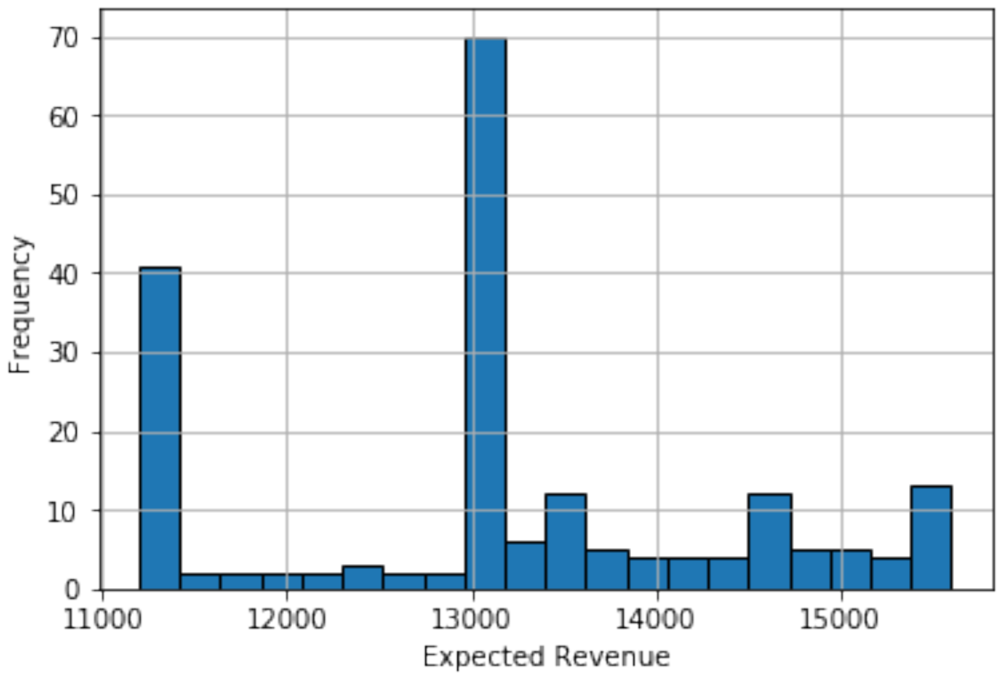
The theory behind this decision rule of whether to accept/reject the low fare customer is based on how many low fare customers we rejected in a row. This factor is mitigated by either accepting a low fare customer or accepting a high fare customer. This is counted by (w) which increases by 1 when a fare is accepted and decreases by 1 when low fare is rejected. We set the criteria to begin accepting low fares when w is less than the benchmark point (x). I.e this means that low fares are recejected at least w times in a row.

```
In [25]: custom_criteria = pd.DataFrame([consec(i) for i in range (-100,100,1)])
         custom_criteria["x"] = np.arange(-100,100)
         custom_criteria.head()
```

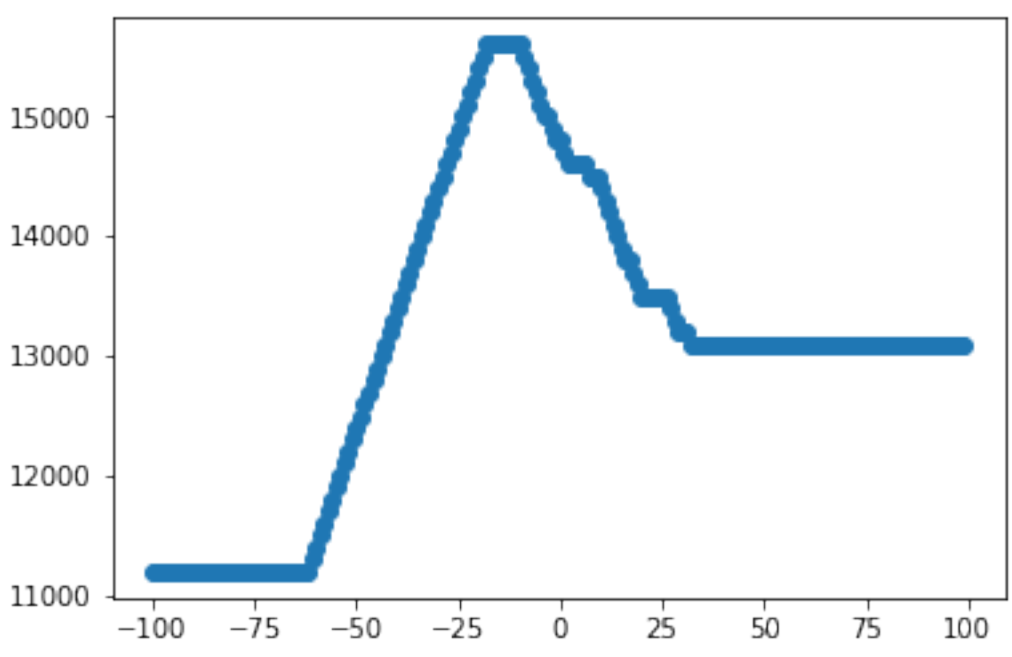
Out[25]:

	0	x
0	11200.0	-100
1	11200.0	-99
2	11200.0	-98
3	11200.0	-97
4	11200.0	-96

```
In [26]: # Distribution of the revenues
         plt.hist(custom_criteria[0], bins = 20, edgecolor ="black")
         plt.xlabel("Expected Revenue")
         plt.ylabel("Frequency")
         plt.grid(True)

         plt.show()
```



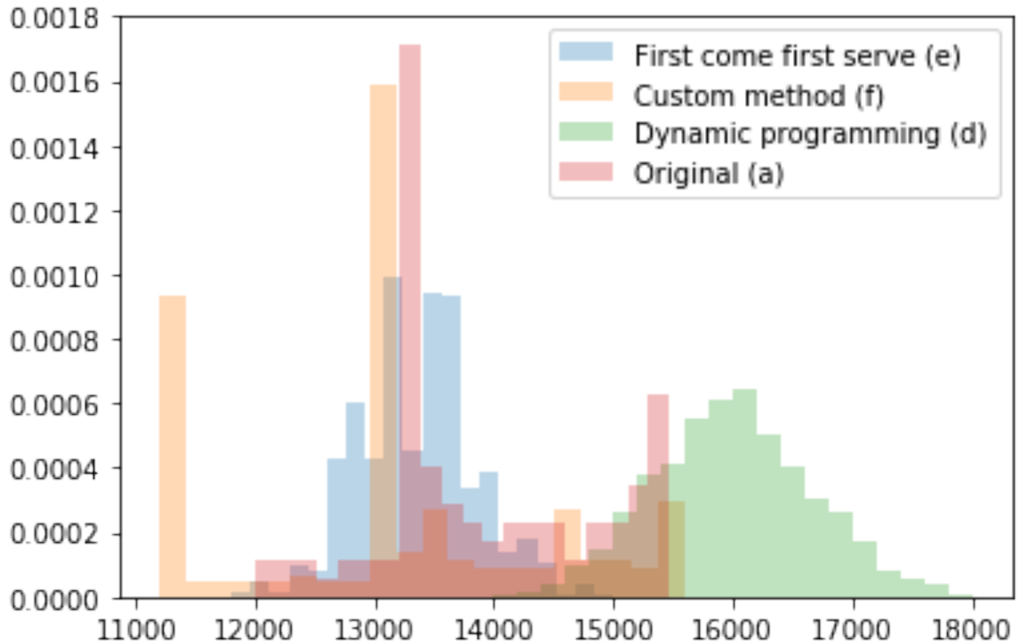

```
In [27]: # Revenue vs Benchmark Plot
         plt.scatter(x=custom_criteria["x"],y=custom_criteria[0])
         plt.show()
```



Distribution Comments

Here we can see that the ditribution of this method has a far reach. While we can get a high revenue, this method may mostlikely produce either a low (12000) or medium (13400) level of revenue. From the plot we can see that the optimal benchmark to alloacte for this case would be around -25.

```
In [28]: # Comparisons of the different methods

         #part_a_rev[0] # part a data a)
         #first_serve # first come frist serve basis e)
         #custom_criteria[0] # custom acceptance criteria f)
         #dyna_rev # dynamic programming d)

         plt.hist(first_serve, label = "First come first serve (e)", histtype = "stepfilled", a
         plt.hist(custom_criteria[0], label = "Custom method (f)",histtype = "stepfilled", alph
         plt.hist(dyna_rev, label ="Dynamic programming (d)",histtype = "stepfilled", alpha = (
         plt.hist(part_a_rev[0], label = "Original (a)",histtype = "stepfilled", alpha = 0.3, c

         plt.legend(loc='upper right')
         plt.show()
```



Methods Comparisons

We can see that the dynamic programming method performs the best, with the highest revenue as well as the highest average. However, one can argue that the Custom Method or the Original Method may offer the greatest stability since the probability of achieving that level of revenue(13400) is the greatest within that method. The Custom Method probably presents the greatest downside with a high probability of getting a very low revenue (12000).