SaasSafras Case Study

Vivek Saravanan

In [33]:
```
# Importing useful packages
import random
import pandas as pd
import deap
import pulp
from deap import base, creator, tools, algorithms
```

In [25]:
```
# Code to ignore warnings - for aesthetics post analysis
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
```

Baselines

To start off the analysis, I wanted to establish some baselines that would serve as a basis to compare against the different allocations. The three baselines I used were:

- Baseline 1: Evenly distributed allocations across roles [7, 7, 6]
- Baseline 2: 20 New Business Acquisition people, none others
- Baseline 3: 20 Account Managers, none others
- Baseline 4: 20 Support Agents, none others

The function below simulates a full year.

In [26]:
```
def get_history(individual):
    history = pd.DataFrame(columns=["month", "customers", "churn", "csat", "nba_members", "am_members", "support_meml
    ORGANIC_ACQUISITION = 25
    BASE_CHURN = 0.1
    BASE_CSAT = 0.7
    PRICE_PER_CUSTOMER = 100
    CUSTOMERS_PER_AM = 25
    n_customers = 1000

    for month, allocation in enumerate(individual, start=1):

        n_acquisition = allocation[0]
        n_account_mgmt = allocation[1]
        n_support = allocation[2]

        # Calculate new acquisitions
        new_acquisitions = n_acquisition * 5 + ORGANIC_ACQUISITION

        # Calculate churn rate
        churn_rate = BASE_CHURN - (BASE_CHURN * ((BASE_CSAT + (n_support * 0.01)) * 0.15))

        # Calculate churn for customers with and without account managers
        n_customers_with_am = min(round(n_account_mgmt) * CUSTOMERS_PER_AM, n_customers)
        n_customers_without_am = max(0, round(n_customers - n_customers_with_am))
        churn_with_am = round(n_customers_with_am * (churn_rate - churn_rate * 0.05))
        churn_without_am = round(n_customers_without_am * churn_rate)

        ## Weighted churn rate
        churn_rate_without_am = churn_rate
        churn_rate_with_am = churn_rate_without_am * 0.95  # Account management reduces churn by 5%

        # Calculate proportion of customers with and without account managers
        proportion_with_am = n_customers_with_am / n_customers
        proportion_without_am = n_customers_without_am / n_customers

        # Weighted churn rate
        weighted_churn_rate = churn_rate_with_am * proportion_with_am + churn_rate_without_am * proportion_without_ar

        # Calculate revenue from customers with and without account managers
        revenue_with_am = n_customers_with_am * PRICE_PER_CUSTOMER * 1.25
        revenue_without_am = n_customers_without_am * PRICE_PER_CUSTOMER

        # Update number of customers
        n_customers = n_customers + new_acquisitions - churn_with_am - churn_without_am

        # Calculate total revenue for the month
        monthly_revenue = revenue_with_am + revenue_without_am
        history = history.append({"month": month, "customers": n_customers, "churn": weighted_churn_rate, "csat": 0.'
                                  "nba_members": n_acquisition, "am_members": n_account_mgmt,
                                  "support_members": n_support, "revenue": monthly_revenue},
                                 ignore_index=True)

    return history
```

In [27]:
```
baseline_1 = [[7, 7, 6], [7, 7, 6], [7, 7, 6], [7, 7, 6], [7, 7, 6], [7, 7, 6], [7, 7, 6], [7, 7, 6], [7, 7, 6], [7,
baseline_2 = [[20, 0, 0], [20, 0, 0], [20, 0, 0], [20, 0, 0], [20, 0, 0], [20, 0, 0], [20, 0, 0], [20, 0, 0], [20, 0
baseline_3 = [[0, 20, 0], [0, 20, 0], [0, 20, 0], [0, 20, 0], [0, 20, 0], [0, 20, 0], [0, 20, 0], [0, 20, 0], [0, 20
baseline_4 = [[0, 0, 20], [0, 0, 20], [0, 0, 20], [0, 0, 20], [0, 0, 20], [0, 0, 20], [0, 0, 20], [0, 0, 20], [0, 0,

print("Baseline 1:", sum(get_history(baseline_1)['revenue']))
print("Baseline 2:", sum(get_history(baseline_2)['revenue']))
print("Baseline 3:", sum(get_history(baseline_3)['revenue']))
print("Baseline 4:", sum(get_history(baseline_4)['revenue']))
```

```
Baseline 1: 1112700.0
Baseline 2: 1375800.0
Baseline 3: 1037700.0
Baseline 4: 890700.0
```

In [42]:
```
get_history(baseline_1)
```

	month	customers	churn	csat	nba_members	am_members	support_members	revenue
0	1.0	972.0	0.087825	0.76	7.0	7.0	6.0	104375.0
1	2.0	946.0	0.087802	0.76	7.0	7.0	6.0	101575.0
2	3.0	923.0	0.08778	0.76	7.0	7.0	6.0	98975.0
3	4.0	902.0	0.08776	0.76	7.0	7.0	6.0	96675.0
4	5.0	883.0	0.087741	0.76	7.0	7.0	6.0	94575.0
5	6.0	865.0	0.087722	0.76	7.0	7.0	6.0	92675.0
6	7.0	849.0	0.087704	0.76	7.0	7.0	6.0	90875.0
7	8.0	834.0	0.087687	0.76	7.0	7.0	6.0	89275.0
8	9.0	821.0	0.08767	0.76	7.0	7.0	6.0	87775.0
9	10.0	809.0	0.087656	0.76	7.0	7.0	6.0	86475.0
10	11.0	798.0	0.087642	0.76	7.0	7.0	6.0	85275.0
11	12.0	788.0	0.087629	0.76	7.0	7.0	6.0	84175.0

Note: The simulation for baseline 1

From these baselines, we can see that the highest revenue results from baseline 2: the scenario when all the employees are assigned to the new business acquisition role. This provides some inclination that the this role might contribute heavily to the overall revenue run rate.

Optimization

Now having some baselines to compare with, I explored some methods I could use to optimize the allocation of employees. A key factor in our situation is that employees can switch roles at the start of every month. Therefore, our optimization would have to optimize each month for an entire year to maximize the run rate revenue.

I researched and explored 2 optimization techniques, starting with linear programming.

- Linear Programming (https://developers.google.com/optimization/lp)
- Genetic Algorithm (https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_quick_guide.htm)

Linear Programming

I used the PuLP (https://coin-or.github.io/pulp/) library to build a simplistic model of the situation provided. The code is provided below.

However, I quickly discovered that while linear programming is a useful technique that could be applied to optimize many situations, it may not be ideal for this situation. The objective function needs to be linear, and support agents and account managers do not have a linear effect. They affect churn which indirectly effects the revenue.

The function below is an example and does not accurately reflect the situation at hand.

In [35]:
```
from pulp import *

# Initialize the problem
prob = LpProblem("ResourceAllocation", LpMaximize)

# Create decision variables
nba = [LpVariable(f'nba_{i}', lowBound=0, cat='Integer') for i in range(12)]
am = [LpVariable(f'am_{i}', lowBound=0, cat='Integer') for i in range(12)]
s = [LpVariable(f's_{i}', lowBound=0, cat='Integer') for i in range(12)]

# Define the objective function
prob += lpSum([500*nba[i] + 400*am[i] + 300*s[i] for i in range(12)])

# Add the constraints
for i in range(12):
    prob += nba[i] + am[i] + s[i] <= 20

# Solve the problem
prob.solve()

# Print the results
for i in range(12):
    print(f'Month {i+1}: New Business Acquisition={value(nba[i])}, Account Management={value(am[i])}, Support={value
```

```
Enumerated nodes:              0
Total iterations:              0
Time (CPU seconds):            0.00
Time (Wallclock seconds):      0.01

Option for printingOptions changed from normal to all
Total time (CPU seconds):      0.00   (Wallclock seconds):      0.02

Month 1: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
Month 2: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
Month 3: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
Month 4: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
Month 5: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
Month 6: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
Month 7: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
Month 8: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
Month 9: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
Month 10: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
Month 11: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
Month 12: New Business Acquisition=20.0, Account Management=0.0, Support=0.0
```

Note: The function above has arbritrary weights for each of the roles, since the effects of these roles cannot be captured linearly

Genetic Algorithm

```
I then explored the [DEAP](https://deap.readthedocs.io/en/master/) library to implement a model that used a genetic
algorithm. This works by finding an optimal solution similar to the process of natural selection and evolution.

I first implemented a few functions that would be used in the optimization
```

Attribute Allocation: This function is the attribute generator. It starts with a base allocation of 3 members in each of the three roles (customer acquisition, account management, and customer support). It then randomly distributes the remaining 11 members among the three roles. The function returns an allocation of team members to roles (a list of three integers).

In [38]:
```
def attr_allocation():
    allocation = [3, 3, 3]  # Start with minimum of 3 in each allocation
    remaining = 11  # Remaining numbers to distribute

    # Distribute the remaining numbers randomly
    for _ in range(remaining):
        idx = random.randint(0, 2)
        allocation[idx] += 1

    return allocation
```

Evaluation: This function is the evaluation or fitness function. It takes an individual (a list of 12 allocations) and calculates the total revenue over a 12-month period based on the given individual. The function models the monthly acquisitions, churn, and revenue based on the allocation of team members to roles, and returns the total revenue as a tuple.

In [39]:
```
def evaluate(individual):
    ORGANIC_ACQUISITION = 25
    BASE_CHURN = 0.1
    BASE_CSAT = 0.7
    PRICE_PER_CUSTOMER = 100
    CUSTOMERS_PER_AM = 25
    n_customers = 1000
    revenue = 0

    for allocation in individual:
        n_acquisition = allocation[0]
        n_account_mgmt = allocation[1]
        n_support = allocation[2]

        # Calculate new acquisitions
        new_acquisitions = n_acquisition * 5 + ORGANIC_ACQUISITION

        # Calculate churn rate
        churn_rate = BASE_CHURN - (BASE_CHURN * ((BASE_CSAT + (n_support * 0.01)) * 0.15))

        # Calculate churn for customers with and without account managers
        n_customers_with_am = min(round(n_account_mgmt * CUSTOMERS_PER_AM), n_customers)
        n_customers_without_am = max(0, round(n_customers - n_customers_with_am))
        churn_with_am = round(n_customers_with_am * (churn_rate - churn_rate * 0.05))
        churn_without_am = round(n_customers_without_am * churn_rate)

        ## Weighted churn rate
        churn_rate_without_am = churn_rate
        churn_rate_with_am = churn_rate_without_am * 0.95  # Account management reduces churn by 5%

        # Calculate proportion of customers with and without account managers
        proportion_with_am = n_customers_with_am / n_customers
        proportion_without_am = n_customers_without_am / n_customers

        # Weighted churn rate
        weighted_churn_rate = churn_rate_with_am * proportion_with_am + churn_rate_without_am * proportion_without_ar

        # Calculate revenue from customers with and without account managers
        revenue_with_am = n_customers_with_am * PRICE_PER_CUSTOMER * 1.25
        revenue_without_am = n_customers_without_am * PRICE_PER_CUSTOMER

        # Update number of customers
        n_customers = n_customers + new_acquisitions - churn_with_am - churn_without_am

        # Calculate total revenue for the month
        monthly_revenue = revenue_with_am + revenue_without_am
        revenue += monthly_revenue

    return revenue,
```

Mutation: This function applies mutation to a given individual. For each month, it randomly selects a role and decreases the number of team members in that role. It then increases the number of team members in another role by the same amount. The function returns the mutated individual.

In [40]:
```
def mutation(individual):
    for i in range(len(individual)):
        if random.random() < 0.1:
            role_to_change = random.randint(0, 2)
            if individual[i][role_to_change] > 1:  # Ensure the count is more than 1
                change = random.randint(1, individual[i][role_to_change])
                individual[i][role_to_change] -= change
                role_to_increase = (role_to_change + random.randint(1, 2)) % 3
                individual[i][role_to_increase] += change
    return individual,
```

Crossover: This function applies crossover to two individuals. For each month, it randomly swaps the allocations of the two individuals. The function returns the two individuals after crossover.

In [ ]:
```
def crossover(ind1, ind2):
    for i in range(len(ind1)):
        if random.random() < 0.5:
            ind1[i], ind2[i] = ind2[i], ind1[i]
    return ind1, ind2
```

Putting it all together: With these functions defined, I continued on to creating the environment in which the genetic algorithm would be run.

Apopulation of 100 individuals is created. Each individual is a list of 12 allocations, representing the allocation of team members to roles over a 12-month period. The script then runs the genetic algorithm for 40 generations, using tournament selection, crossover, and mutation. Finally, the script prints the best individual and its total revenue, and generates a history DataFrame for the best individual.

In [36]:
```
random.seed(2)
# Set up the fitness and individual classes
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

# Create the toolbox
toolbox = base.Toolbox()

toolbox.register("attr_allocation", attr_allocation)

# Structure initializers
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_allocation, 12)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Set up the genetic operators
toolbox.register("evaluate", evaluate)
toolbox.register("mate", crossover)  # Updated crossover
toolbox.register("mutate", mutation)  # Updated mutation
toolbox.register("select", tools.selTournament, tournsize=3)

# Set up the population
pop = toolbox.population(n=100)

# Run the genetic algorithm
result, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=40, verbose=False)

# Print the best individual
best_individual = tools.selBest(result, k=1)[0]
print('Best Individual: ', best_individual, 'Revenue: ', evaluate(best_individual)[0])

# Generate the history DataFrame for the best individual
history = get_history(best_individual)
print("Monthly metrics for the best individual:")
history
```

0	1.0	1035.0	0.0895	0.7	20.0	0.0	0.0	100000.0
1	2.0	1053.0	0.089134	0.71	17.0	2.0	1.0	104750.0
2	3.0	1074.0	0.0892	0.72	18.0	0.0	2.0	105300.0
3	4.0	1098.0	0.089396	0.7	19.0	1.0	0.0	108025.0
4	5.0	1125.0	0.0895	0.7	20.0	0.0	0.0	109800.0
5	6.0	1144.0	0.08935	0.71	19.0	0.0	1.0	112500.0
6	7.0	1147.0	0.089005	0.72	16.0	2.0	2.0	115650.0
7	8.0	1160.0	0.089253	0.71	18.0	1.0	1.0	115325.0
8	9.0	1181.0	0.0895	0.7	20.0	0.0	0.0	116000.0
9	10.0	1151.0	0.088445	0.72	10.0	8.0	2.0	123100.0
10	11.0	1125.0	0.088215	0.76	10.0	4.0	6.0	117600.0
11	12.0	1067.0	0.087616	0.74	3.0	13.0	4.0	120625.0

In [43]:
```
sum(history['revenue'])
```

Out[43]: 1348675.0

Note: The "Best Individual" list shows the optimal number of team members allocated to each of the three roles (Customer Acquisition, Account Management, and Customer Support) for each month in a year.

Using the genetic algorithm, I found an optimal allocation of employees for each role with the focus of maximizing revenue. The GA found a good solution (i.e., it significantly increased the revenue), but it didn't find the best possible solution as the the revenue from this was lower than Baseline 2. However, the genetic algorithm is just a heuristic and approaches the optimal solution, so this was not a failure.

Interpretation: This resource allocation strategy only focused on maximizing revenue. It can be interpreted as an aggressive acquisition phase in the early months, gradually transitioning to a more balanced approach with increased focus on customer retention and support as the customer base grows.

Adjustments

The GA strategy and algorithm provided an optimization for the maximum revenue, that provided an output of allocations that at times allocated zero resources to a single role. Additionally, in reality, the cost of acquiring a new customer is often more expensive than retaining one. Factoring this in, I made an adjustment to the genetic algorithm, to primarily maximize revenue, but also minimize churn as a secondary objective.

Evaluation: This new evaluation function also calculates the overall average churn rate and aims to minimize this over the course of the year

In [44]:
```
def evaluate(individual):
    ORGANIC_ACQUISITION = 25
    BASE_CHURN = 0.1
    BASE_CSAT = 0.7
    PRICE_PER_CUSTOMER = 100
    CUSTOMERS_PER_AM = 25
    n_customers = 1000
    revenue = 0
    total_churn_rate = 0 #newline

    for allocation in individual:
        n_acquisition = allocation[0]
        n_account_mgmt = allocation[1]
        n_support = allocation[2]

        # Calculate new acquisitions
        new_acquisitions = n_acquisition * 5 + ORGANIC_ACQUISITION

        # Calculate churn rate
        churn_rate = BASE_CHURN - (BASE_CHURN * ((BASE_CSAT + (n_support * 0.01)) * 0.15))

        # Calculate churn for customers with and without account managers
        n_customers_with_am = min(round(n_account_mgmt * CUSTOMERS_PER_AM), n_customers)
        n_customers_without_am = max(0, n_customers - round(n_customers_with_am))
        churn_with_am = round(n_customers_with_am * (churn_rate - churn_rate * 0.05))
        churn_without_am = round(n_customers_without_am * churn_rate)

        ## Weighted churn rate
        churn_rate_without_am = churn_rate
        churn_rate_with_am = churn_rate_without_am * 0.95  # Account management reduces churn by 5%

        # Calculate proportion of customers with and without account managers
        proportion_with_am = n_customers_with_am / n_customers
        proportion_without_am = n_customers_without_am / n_customers

        # Weighted churn rate
        weighted_churn_rate = churn_rate_with_am * proportion_with_am + churn_rate_without_am * proportion_without_ar

        # Calculate revenue from customers with and without account managers
        revenue_with_am = n_customers_with_am * PRICE_PER_CUSTOMER * 1.25
        revenue_without_am = n_customers_without_am * PRICE_PER_CUSTOMER

        # Update number of customers
        n_customers = n_customers + new_acquisitions - churn_with_am - churn_without_am

        # Calculate total revenue for the month
        monthly_revenue = revenue_with_am + revenue_without_am
        revenue += monthly_revenue
        total_churn_rate += weighted_churn_rate #newline

    average_churn_rate = total_churn_rate / len(individual)  # calculate average churn rate
    return revenue, average_churn_rate #added total_churn
```

A few tweaks to the environment setup. The objective is FitnessMulti with 2 weights that correspond to revenue and churn. I also updated the selection method from tournament to NSGA-II, more suitable for this task

In [57]:
```
random.seed(12)

# Set up the fitness and individual classes
creator.create("FitnessMulti", base.Fitness, weights=(1.0, -0.5)) #added new weight and changed to fitnessmulti
creator.create("Individual", list, fitness=creator.FitnessMulti) #changed to fitness multi

# Create the toolbox
toolbox = base.Toolbox()

toolbox.register("attr_allocation", attr_allocation)

# Structure initializers
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_allocation, 12)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Set up the genetic operators
toolbox.register("evaluate", evaluate)
toolbox.register("mate", crossover)  # Updated crossover
toolbox.register("mutate", mutation)  # Updated mutation
toolbox.register("select", tools.selNSGA2)  # Use NSGA-II for selection

# Set up the population
pop = toolbox.population(n=100)

# Run the genetic algorithm
result, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=40, verbose=False)

# Print the best individual
best_individual = tools.selNSGA2(result, k=1)[0]
print('Best Individual: ', best_individual, 'Revenue: ', evaluate(best_individual)[0])

# Generate the history DataFrame for the best individual
history = get_history(best_individual)
print("Monthly metrics for the best individual:")
history
```

0	1.0	973.0	0.087748	0.78	7.0	5.0	8.0	103125.0
1	2.0	997.0	0.08912	0.71	17.0	2.0	1.0	98550.0
2	3.0	994.0	0.088416	0.75	12.0	3.0	5.0	101575.0
3	4.0	966.0	0.08782	0.76	7.0	7.0	6.0	103775.0
4	5.0	951.0	0.088095	0.74	9.0	7.0	4.0	100975.0
5	6.0	933.0	0.088058	0.71	8.0	11.0	1.0	101975.0
6	7.0	916.0	0.087947	0.74	8.0	8.0	4.0	98300.0
7	8.0	940.0	0.088928	0.73	16.0	1.0	3.0	92225.0
8	9.0	922.0	0.087893	0.76	8.0	6.0	6.0	97750.0
9	10.0	892.0	0.087343	0.82	5.0	3.0	12.0	94075.0
10	11.0	878.0	0.087995	0.7	8.0	12.0	0.0	96700.0
11	12.0	841.0	0.087334	0.7	3.0	17.0	0.0	98425.0

Interpretation:

1. *New Business Acquisition*: In the first month, there are 7 staff members allocated to NBA, but the number changes throughout the year, reaching a peak of 17 in the second month and a low of 3 in the last month. This could indicate that focusing more resources on acquiring new customers early in the year may bring in more revenue. In the last two months, the number decreases significantly, which could be due to various reasons like anticipating less business during the end of the year or reallocating resources to other departments.
2. *Account Management*: The number of staff in AM also changes monthly, starting from 5 in the first month and peaking at 17 in the last month. This might suggest that maintaining relationships with existing customers (which AM focuses on) becomes more important over time. A possible reason could be that as the customer base grows, it's essential to invest more in keeping these customers satisfied and reduce churn, which can be costlier than acquiring new customers.
3. *Customer Support*: The number of CS staff starts at 8 and fluctuates throughout the year, reaching a peak of 12 in the tenth month. This might suggest that, particularly during certain times, it's crucial to provide excellent customer service to maintain a high level of customer satisfaction, reducing churn and therefore maintaining revenue.

The total revenue generated using this allocation strategy is $1,187,450. This also falls short of our baseline, but also accounts for the minimization of churn.

It's important to note that these allocations are based on the specific parameters and assumptions defined in your model, and they might change with different parameters or in real-world situations.

The key pattern seems to be a balancing act between acquiring new customers, maintaining relationships, and providing excellent customer service. In general, the algorithm seems to be suggesting that shifting more resources to Account Managers and Customer Support as the year progresses, while starting the year strong with New Business Acquisition.

Using these insights, I reallocated the distributions keeping in mind team dynamics

In [61]:
```
history = get_history([[12,4,4], [12,4,4], [11,5,4], [10,6,4], [9,6,5], [9,7,4], [7,8,5], [7,8,5], [6,9,5], [5,9,6],
history
```

Out[61]:

	month	customers	churn	csat	nba_members	am_members	support_members	revenue
0	1.0	997.0	0.088456	0.74	12.0	4.0	4.0	102500.0
1	2.0	994.0	0.088454	0.74	12.0	4.0	4.0	102200.0
2	3.0	986.0	0.088341	0.74	11.0	5.0	4.0	102525.0
3	4.0	974.0	0.088224	0.74	10.0	6.0	4.0	102350.0
4	5.0	958.0	0.088067	0.75	9.0	6.0	5.0	101150.0
5	6.0	943.0	0.088088	0.74	9.0	7.0	4.0	100175.0
6	7.0	920.0	0.087809	0.75	7.0	8.0	5.0	99300.0
7	8.0	899.0	0.087785	0.75	7.0	8.0	5.0	97000.0
8	9.0	875.0	0.087639	0.75	6.0	9.0	5.0	95525.0
9	10.0	848.0	0.087461	0.76	5.0	9.0	6.0	93125.0
10	11.0	819.0	0.087294	0.76	4.0	10.0	6.0	91050.0
11	12.0	793.0	0.08726	0.75	4.0	11.0	5.0	88775.0

In [62]:
```
sum(history['revenue'])
```

Out[62]: 1175675.0