Cover page feather background with text block

**Introduction**

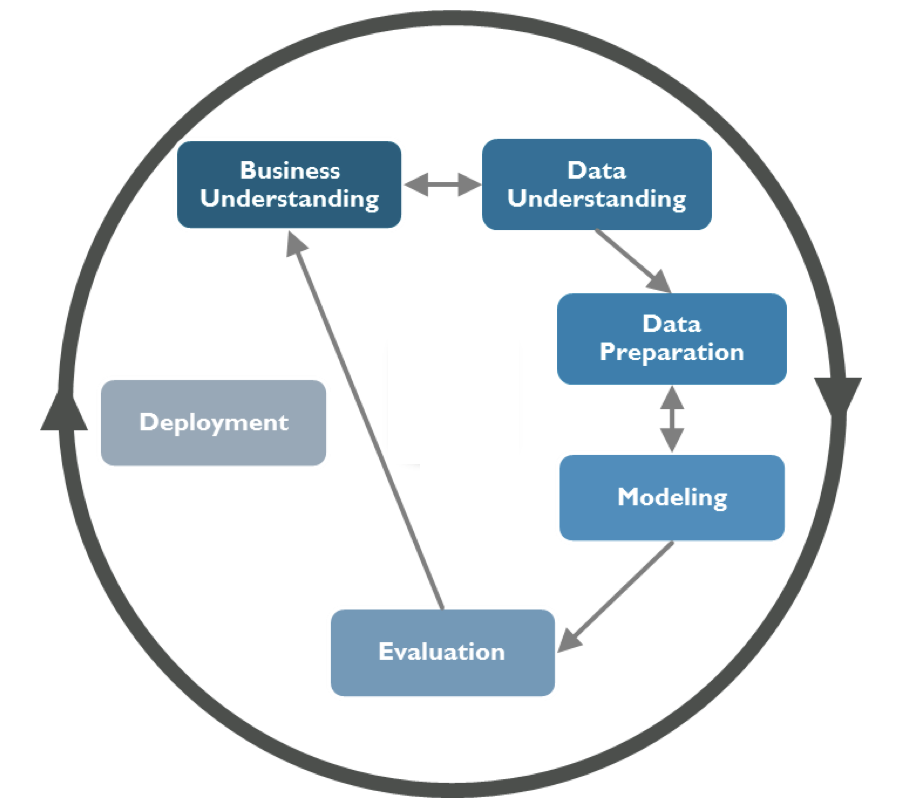
Brainstorming is an important part of every decision. Studies show that 36% of conversations contain factual errors which can lead to incorrect decisions. Additionally, individuals can also draw a blank, or not know how to proceed from certain roadblocks. This is especially occurrent in business decisions, where people can make decisions without taking the proper considerations into mind such as market research and cost analysis. We attempt to make a brainstorming tool to help with exactly this problem. Users will be able to ask questions and the program will guide them to a proper decision. The responses will be personalized to the user’s location, business, product, and target market in order to ensure that the user will be making the correct decisions for their company. This program will provide the user will factually correct information to help them reach their decision.

**PROBLEM STATEMENT**

90% of startups fail due to preventable reasons such as not meeting a market need, scaling up to early, or poor management.This project will assist users and potential business owners in making correct decisions.

**SCOPE OF THE PROJECT**

This project will be specialized for business use. The complete project can handle questions from marketing, finance, and resource management. However, the current program is equipped to handle finance questions regarding what the user’s should price their product. This product can be used by almost any type of business for every target market. The program also cannot function without inputted information. For example, the program currently cannot suggest a price of a product, it can only confirm/deny a proposed price. Due to time constraints, the program does not have a User Interface.



CRISP DIAGRAM



FLOWCHART FOR IMPLEMENTATION

IMPLEMENTATION

3.1 Data Collection

We used web scraping tools to get data from a variety of sources like Craft.co, Y-combinators. The scraper was run on Portia. Once we got the data, we stored it as csv format. We faced a lot of problems while extracting data from websites. We are looking to make this process more streamlined and auto update the company lists and the related information. We gathered demographic information from the various city based websites provided by the census and used it accordingly in our website.

3.2 Pre-processor for training the data:

We made a database to store the CSV data from the collected data We then stored the details in different dictionaries (Python dictionary) after extracting it from the database. This data in the dictionaries were initially cleaned to remove unwanted noise from the text.

3.3 Training the chatterbot:

The chatterbot was designed to find similarity between the description entered by the user and the existing descriptions in the dataset of companies. The similarity score was determined for each and every description and based on the decided threshold, we then suggested these companies as primary competitions to the user’s product. This would give them all the information on the competitions and their corresponding values. We would in future make the list an exhaustive list for better and detailed list of competitive companies.

3.4 Determining the viability of the Product:

Based on the other inputs of the user (like product cost, product use frequency) we then trained an algorithm to decide whether the product is viable or not. We used Logistic regression model to predict whether pricing of the product is viable or not. This from the user was collected through the prompt and the data was then added to the database for computational purposes.

CHALLENGES

One of the biggest challenges was getting large amounts of data on demographics, income levels, and marketing/finance strategies. This took the majority of the time.

Another issue we ran into was that the open source framework (Chatterbot) that we worked with didn’t have a lot of documentation.The difficulty in creating custom functions to modify its usage according to our needs was compounded by the fact that there was little documentation.

Additionally, time was a big issue as we had a big plans with this project and due to the constraints on we didn’t meet our entire list of goals.

## Conclusion

This project can prove to be useful to many business owners who are inexperienced in the daily decisions required to run a company. With all the features added, this brainstorming tool can help them constructively think about ideas and implementations in marketing, finance and resource management. This program will introduce considerations that the user should think about while making decisions, as well as lead them to the correct choice for their company. Future uses of this product include opening up to educational institutions to help students in their brainstorming for essays and project ideas.

Appendix-A

**Comparisons.py**

|  |
| --- |
| """ This module contains various text-comparison algorithms designed to compare one statement to another. """ from chatterbot import utils from chatterbot import languages from chatterbot import tokenizers from nltk.corpus import wordnet, stopwords  # Use python-Levenshtein if available try:  from Levenshtein.StringMatcher import StringMatcher as SequenceMatcher except ImportError:  from difflib import SequenceMatcher   class Comparator:   def \_\_call\_\_(self, statement\_a, statement\_b):  return self.compare(statement\_a, statement\_b)   def compare(self, statement\_a, statement\_b):  return 0   class LevenshteinDistance(Comparator):  """  Compare two statements based on the Levenshtein distance  of each statement's text.   For example, there is a 65% similarity between the statements  "where is the post office?" and "looking for the post office"  based on the Levenshtein distance algorithm.  """   def compare(self, statement, other\_statement):  """  Compare the two input statements.   :return: The percent of similarity between the text of the statements.  :rtype: float  """   # Return 0 if either statement has a falsy text value  if not statement or not other\_statement:  return 0   # Get the lowercase version of both strings  statement\_text = str(statement.lower())  other\_statement\_text = str(other\_statement.lower())   similarity = SequenceMatcher(  None,  statement\_text,  other\_statement\_text  )   # Calculate a decimal percent of the similarity  percent = round(similarity.ratio(), 2)   return percent   class SynsetDistance(Comparator):  """  Calculate the similarity of two statements.  This is based on the total maximum synset similarity between each word in each sentence.   This algorithm uses the `wordnet`\_ functionality of `NLTK`\_ to determine the similarity  of two statements based on the path similarity between each token of each statement.  This is essentially an evaluation of the closeness of synonyms.  """   def \_\_init\_\_(self):  super().\_\_init\_\_()   self.language = languages.ENG   self.stopwords = None   self.word\_tokenizer = None   self.initialization\_functions = [  utils.download\_nltk\_wordnet,  utils.download\_nltk\_stopwords  ]   def get\_stopwords(self):  """  Get the list of stopwords from the NLTK corpus.  """  if self.stopwords is None:  self.stopwords = stopwords.words(self.language.ENGLISH\_NAME.lower())   return self.stopwords   def get\_word\_tokenizer(self):  """  Get the word tokenizer for this comparison algorithm.  """  if self.word\_tokenizer is None:  self.word\_tokenizer = tokenizers.get\_word\_tokenizer(self.language)   return self.word\_tokenizer   def compare(self, statement, other\_statement):  """  Compare the two input statements.   :return: The percent of similarity between the closest synset distance.  :rtype: float   .. \_wordnet: http://www.nltk.org/howto/wordnet.html  .. \_NLTK: http://www.nltk.org/  """  import itertools   word\_tokenizer = self.get\_word\_tokenizer()   tokens1 = word\_tokenizer.tokenize(statement.text.lower())  tokens2 = word\_tokenizer.tokenize(other\_statement.text.lower())   # Get the stopwords for the current language  stop\_word\_set = set(self.get\_stopwords())   # Remove all stop words from the list of word tokens  tokens1 = set(tokens1) - stop\_word\_set  tokens2 = set(tokens2) - stop\_word\_set   # The maximum possible similarity is an exact match  # Because path\_similarity returns a value between 0 and 1,  # max\_possible\_similarity is the number of words in the longer  # of the two input statements.  max\_possible\_similarity = min(  len(tokens1),  len(tokens2)  ) / max(  len(tokens1),  len(tokens2)  )   max\_similarity = 0.0   # Get the highest matching value for each possible combination of words  for combination in itertools.product(\*[tokens1, tokens2]):   synset1 = wordnet.synsets(combination[0])  synset2 = wordnet.synsets(combination[1])   if synset1 and synset2:   # Get the highest similarity for each combination of synsets  for synset in itertools.product(\*[synset1, synset2]):  similarity = synset[0].path\_similarity(synset[1])   if similarity and (similarity > max\_similarity):  max\_similarity = similarity   if max\_possible\_similarity == 0:  return 0   return max\_similarity / max\_possible\_similarity   class SentimentComparison(Comparator):  """  Calculate the similarity of two statements based on the closeness of  the sentiment value calculated for each statement.  """   def \_\_init\_\_(self):  super().\_\_init\_\_()   self.sentiment\_analyzer = None   self.initialization\_functions = [  utils.download\_nltk\_vader\_lexicon  ]   def get\_sentiment\_analyzer(self):  """  Get the initialized sentiment analyzer.  """  if self.sentiment\_analyzer is None:  from nltk.sentiment.vader import SentimentIntensityAnalyzer   self.sentiment\_analyzer = SentimentIntensityAnalyzer()   return self.sentiment\_analyzer   def compare(self, statement, other\_statement):  """  Return the similarity of two statements based on  their calculated sentiment values.   :return: The percent of similarity between the sentiment value.  :rtype: float  """  sentiment\_analyzer = self.get\_sentiment\_analyzer()  statement\_polarity = sentiment\_analyzer.polarity\_scores(statement.text.lower())  statement2\_polarity = sentiment\_analyzer.polarity\_scores(other\_statement.text.lower())   statement\_greatest\_polarity = 'neu'  statement\_greatest\_score = -1  for polarity in sorted(statement\_polarity):  if statement\_polarity[polarity] > statement\_greatest\_score:  statement\_greatest\_polarity = polarity  statement\_greatest\_score = statement\_polarity[polarity]   statement2\_greatest\_polarity = 'neu'  statement2\_greatest\_score = -1  for polarity in sorted(statement2\_polarity):  if statement2\_polarity[polarity] > statement2\_greatest\_score:  statement2\_greatest\_polarity = polarity  statement2\_greatest\_score = statement2\_polarity[polarity]   # Check if the polarity if of a different type  if statement\_greatest\_polarity != statement2\_greatest\_polarity:  return 0   values = [statement\_greatest\_score, statement2\_greatest\_score]  difference = max(values) - min(values)   return 1.0 - difference   class JaccardSimilarity(Comparator):  """  Calculates the similarity of two statements based on the Jaccard index.   The Jaccard index is composed of a numerator and denominator.  In the numerator, we count the number of items that are shared between the sets.  In the denominator, we count the total number of items across both sets.  Let's say we define sentences to be equivalent if 50% or more of their tokens are equivalent.  Here are two sample sentences:   The young cat is hungry.  The cat is very hungry.   When we parse these sentences to remove stopwords, we end up with the following two sets:   {young, cat, hungry}  {cat, very, hungry}   In our example above, our intersection is {cat, hungry}, which has count of two.  The union of the sets is {young, cat, very, hungry}, which has a count of four.  Therefore, our `Jaccard similarity index`\_ is two divided by four, or 50%.  Given our similarity threshold above, we would consider this to be a match.   .. \_`Jaccard similarity index`: https://en.wikipedia.org/wiki/Jaccard\_index  """   def \_\_init\_\_(self):  super().\_\_init\_\_()   import string   self.punctuation\_table = str.maketrans(dict.fromkeys(string.punctuation))   self.language = languages.ENG   self.stopwords = None   self.lemmatizer = None   self.word\_tokenizer = None   self.initialization\_functions = [  utils.download\_nltk\_wordnet,  utils.download\_nltk\_averaged\_perceptron\_tagger,  utils.download\_nltk\_stopwords  ]   def get\_stopwords(self):  """  Get the list of stopwords from the NLTK corpus.  """  if self.stopwords is None:  self.stopwords = stopwords.words(self.language.ENGLISH\_NAME.lower())   return self.stopwords   def get\_lemmatizer(self):  """  Get the lemmatizer.  """  if self.lemmatizer is None:  from nltk.stem.wordnet import WordNetLemmatizer   self.lemmatizer = WordNetLemmatizer()   return self.lemmatizer   def get\_word\_tokenizer(self):  """  Get the word tokenizer for this comparison algorithm.  """  if self.word\_tokenizer is None:  self.word\_tokenizer = tokenizers.get\_word\_tokenizer(self.language)   return self.word\_tokenizer   def compare(self, statement, other\_statement):  """  Return the calculated similarity of two  statements based on the Jaccard index.  """  from nltk import pos\_tag   word\_tokenizer = self.get\_word\_tokenizer()   # Get the stopwords for the current language  stopwords = self.get\_stopwords()   lemmatizer = self.get\_lemmatizer()   # Make both strings lowercase  a = statement.text.lower()  b = other\_statement.text.lower()   # Remove punctuation from each string  a = a.translate(self.punctuation\_table)  b = b.translate(self.punctuation\_table)   pos\_a = pos\_tag(word\_tokenizer.tokenize(a))  pos\_b = pos\_tag(word\_tokenizer.tokenize(b))   lemma\_a = [  lemmatizer.lemmatize(  token, utils.treebank\_to\_wordnet(pos)  ) for token, pos in pos\_a if token not in stopwords  ]  lemma\_b = [  lemmatizer.lemmatize(  token, utils.treebank\_to\_wordnet(pos)  ) for token, pos in pos\_b if token not in stopwords  ]   # Calculate Jaccard similarity  numerator = len(set(lemma\_a).intersection(lemma\_b))  denominator = float(len(set(lemma\_a).union(lemma\_b)))  ratio = numerator / denominator   return ratio   # ---------------------------------------- #   levenshtein\_distance = LevenshteinDistance() synset\_distance = SynsetDistance() sentiment\_comparison = SentimentComparison() jaccard\_similarity = JaccardSimilarity() |

**chatbot.py**

|  |
| --- |
| from chatterbot import ChatBot from chatterbot.trainers import ListTrainer  # Create a new chat bot named Charlie chatbot = ChatBot('Charlie')  trainer = ListTrainer(chatbot)  trainer.train([  "Hi, can I help you?",  "Sure, I'd like to book a flight to Iceland.",  "Your flight has been booked." ])  # Get a response to the input text 'I would like to book a flight.' response = chatbot.get\_response('I would like to book a flight.')  print(response) |

**chatbottest.py**

|  |
| --- |
| from chatterbot.comparisons import levenshtein\_distance import csv  columns = [] with open('Startups.csv', encoding="utf8") as f:  reader = csv.reader(f)  for row in reader:  if columns:  for i, value in enumerate(row):  columns[i].append(value)  else:  # first row  columns = [[value] for value in row] # you now have a column-major 2D array of your file. as\_dict = {c[0] : c[1:] for c in columns} #print(as\_dict)   descriptions = as\_dict['Description']  #print(descriptions) companies = as\_dict['Company'] median\_household\_income = 48000   class UserDetails(object):  def \_\_init\_\_(self):  self.product\_description = ""  self.similar\_companies = []  self.product\_location = ""  self.age\_group = 0  self.product\_cost = 0  self.product\_category = ""  self.product\_frequency = ""   def print\_data(self):  print("Product Description: " + self.product\_description)  print("List of similar companies: ")  for i in range(len(self.similar\_companies)):  print(self.similar\_companies[i])  print("Product Target Location: "+ self.product\_location)  print("Product Target Age group: "+ self.age\_group)  print("Product Cost: " + str(self.product\_cost))  print("Product Category: " + self.product\_category)  print("Product Usage Frequency: " + self.product\_frequency)   profile = UserDetails()  category\_weights = {"Electronics": 10,  "Subscriptions": 10,  "Transportation": 10,  "Housing": 20,  "Education": 10,  "Clothing": 10,  "Food": 30}   product\_frequency = {"One-time": 1,  "Scarcely": 12,  "Occasionally": 100,  "Frequently": 200,  "Everyday": 365}   while True:  print("Enter your company's description")  userInput = input("You: ")  if userInput.strip() == 'Bye' or userInput.strip() == 'bye':  print('ChatBot: Bye')  break  else:  for i in range(len(descriptions)):  #print(descriptions[i])  #print(levenshtein\_distance.compare(userInput, descriptions[i]))  if levenshtein\_distance.compare(userInput, descriptions[i]) > 0.45:  #print(companies[i])  profile.similar\_companies.append(companies[i])  profile.product\_description = userInput   print("Enter your target location")  userInput = input("You: ")  if userInput.strip() == 'Bye' or userInput.strip() == 'bye':  print("ChatBot: Bye")  break  else:  profile.product\_location = userInput   print("Enter age group of your target market")  print("1. Teenagers (>13 & <20)\n2. Young Adults (>20 & <30)\n3. Adults (>30 & <65)\n4. Seniors (>65)")  userInput = input("You: ")  if userInput.strip() == 'Bye' or userInput.strip() == 'bye':  print("ChatBot: Bye")  break  elif userInput == "1":  profile.age\_group = "Teenagers (>13 & <20)"  elif userInput == "2":  profile.age\_group = "Young Adults (>20 & <30)"  elif userInput == "3":  profile.age\_group = "Adults (>30 & <65)"  elif userInput == "4":  profile.age\_group = "Seniors (>65)"   print("How often is your product going to be used?")  print("1. Scarcely\n2. Occasionally\n3. Frequently\n4. Everyday")  userInput = input("You: ")  if userInput.strip() == 'Bye' or userInput.strip() == 'bye':  print("ChatBot: Bye")  break  elif userInput == "1":  profile.product\_frequency = "One-Time"  elif userInput == "2":  profile.product\_frequency = "Scarcely"  elif userInput == "3":  profile.product\_frequency = "Occasionally"  elif userInput == "4":  profile.product\_frequency = "Frequently"  elif userInput == "5":  profile.product\_frequency = "Everyday"   print("What category does your product fall under?")  print("1. Electronics\n2. Subscriptions\n3. Transportation\n4. Housing\n5. Education\n6. Clothing\n7. Food")  userInput = input("You: ")  if userInput.strip() == 'Bye' or userInput.strip() == 'bye':  print("ChatBot: Bye")  break  elif userInput == "1":  profile.product\_category = "Electronics"  elif userInput == "2":  profile.product\_category = "Subscriptions"  elif userInput == "3":  profile.product\_category = "Transportation"  elif userInput == "4":  profile.product\_category = "Housing"  elif userInput == "5":  profile.product\_category = "Education"  elif userInput == "6":  profile.product\_category = "Clothing"  elif userInput == "7":  profile.product\_category = "Food"   print("Enter your product cost:")  userInput = input("You: ")  if userInput.strip() == 'Bye' or userInput.strip() == 'bye':  print("ChatBot: Bye")  break  else:  profile.product\_cost = int(userInput)   profile.print\_data()  product\_value = profile.product\_cost \* product\_frequency[profile.product\_frequency]  expendable\_income\_for\_category = median\_household\_income \* category\_weights[profile.product\_category] / 100   product\_percent\_of\_income = product\_value / expendable\_income\_for\_category  if product\_percent\_of\_income < 0.1:  print("Viable product")  else:  print("Non-viable product") |

**webscraper.py**

|  |
| --- |
| from bs4 import BeautifulSoup  import requests  url = input("Enter a website to extract the URL's from: ")  r = requests.get("http://" + url)  data = r.text  soup = BeautifulSoup(data, features="html5lib")  for link in soup.find\_all('a'):  print(link.get('href')) |

**csvtodict.py**

|  |
| --- |
| import csv  columns = [] with open('Startups.csv', encoding="utf8") as f:  reader = csv.reader(f)  for row in reader:  if columns:  for i, value in enumerate(row):  columns[i].append(value)  else:  # first row  columns = [[value] for value in row] # you now have a column-major 2D array of your file. as\_dict = {c[0] : c[1:] for c in columns} print(as\_dict) |

OUTPUT

