In [1]:
```
__author__ = "Bruce Pannaman"

import warnings
from IPython.display import display, HTML
warnings.simplefilter('ignore')
display(HTML(data="""<style> div#notebook-container { width: 95%; } </style>
```

Before running this script please do the following steps:

1. pip install -r requirements.txt
2. python3 -m spacy download en_core_web_sm
3. brew install enchant

Initial thoughts

After reading the document, my understanding is that the following functionality is needed:

1. Categorise each short string or phrase into the following categories.
    - Company Names
    - Company Addresses
    - Serial Numbers
    - Physical Goods
    - Locations
2. Establish links between each string within each category to ascertain if they represent the same entity
3. Group and concatonate the entities within each category

Questions for the wider Vector.ai team

Priorities:

1. Who are the primary and subsequent users of the output of this engine?
2. Is accuracy more important than speed?
3. Is there a particilar format/ storage option that this engine can be outputted into e.g. API endpoint, DB entry or module to be used in wider codebase.

Assumptions to be questioned:

1. Are all of these documents in English?
2. How have these stings been obtained and is there a chance of underlying errors e.g. have these strings been taken from scanned images using OCR techniques?
3. For it's use case within vector.ai, should the model focus on precision or recall when grouping entities?

Calculating Locations

To recognise locations, Spacy has a really good named entity model that will give us an accurate reading as to whether the string contains a location. This model will return a result of "LOC" or "GPE" denoting Location or Geopolitical entity.

In [2]:

```
import spacy
# python3 -m spacy download en_core_web_sm
nlp = spacy.load("en_core_web_sm")
```

To gain more granularity on the location once we have established that there is a location or address in the field. There is a library called geograpy3 that can do the heavy lifting of this for us to extract more data from the location string.

In [3]:

```
import geograpy
# This library uses the NLTK library and the following models to operate.
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('maxent_ne_chunker')
nltk.download('words')
```

```
[nltk_data] Downloading package stopwords to /Users/bruce/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /Users/bruce/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /Users/bruce/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package maxent_ne_chunker to
[nltk_data]     /Users/bruce/nltk_data...
[nltk_data]   Package maxent_ne_chunker is already up-to-date!
[nltk_data] Downloading package words to /Users/bruce/nltk_data...
[nltk_data]   Package words is already up-to-date!
```

Out[3]: True

If there is a postcode or zipcode, a lot of data can be extracted from just this part. There is another library called pgeocode, which can extract more data from a postcode or zipcode if the country can be given. I will use regex to extract the postcode/ zipcode and run it against the pgeocode library for my primary location information

In [4]:

```
import re
import pgeocode
```

In [5]:

```
example_locations = ["SLOUGH, SE12 2XY", "33 TIMBER YARD, LONDON, L1 8XY", "4
```

In [6]:

```
class LocationEngine:
    def __init__(self, search_string):
        self.search_string = search_string
        self.contains_locations = False

        # These will be the granularity lists to be returned from the engine
        self.location_contexts = ["countries", "regions"]
        # Create lists to collect each location type
        for location_context in self.location_contexts:
            exec("self.{LC}_list = []".format(LC=location_context))
```

```
    def clean_string(self, string):
        """Cleans the string up to allow for better NLTK operations"""
        return string.strip().lower()

    def locations_found(self):
        """Returns if there is a location found within the string"""
        found_locations = []
        doc = nlp(self.search_string)

        for ent in doc.ents:
            if ent.label_ in ["LOC", "GPE"]:
                found_locations.append(ent.text)

        return len(found_locations) > 0, found_locations

    def check_zip_code_regex(self, string):
        """
        Checks the string against a dict of world zipcode array to establish
        there is a match for an address identifier
        """
        # More can be added for all 195 countries in the world if needed
        postcode_regex_dict = {"gb": "([Gg][Ii][Rr] 0[Aa]{2})|((([A-Za-z][0-9
        # Search each countries address identifier's regex to find a match
        for country_code in postcode_regex_dict:
            result = re.findall(postcode_regex_dict[country_code], string)
            if len(result) > 0:
                return country_code
        return None

    def lookup_zipcode(self, country_code, zipcode):
        """Uses the pgeocode library to gather further information about the
        db = pgeocode.Nominatim(country_code)
        result = db.query_postal_code(zipcode)
        return result.country_code, result.state_name

    def identify_location(self):
        """
        Goes through each line of the string to search for clues as to whethe
        ASSUMPTION - locations & addresses are comma delimited

        """
        # Reverse the list as the strongest signals of a location are at the
        locations_found, location_strings = self.locations_found()

        self.places_list = location_strings
        self.contains_locations = locations_found
        for line in self.search_string.split(",")[::-1]:
            cleaned_string = self.clean_string(line)

            # Check if the substring has a match with a zipcode or postcode f
            zip_code_country = self.check_zip_code_regex(cleaned_string)
            # If there is a match with the zipcode regex, focus in on that in
            if zip_code_country is not None:
                self.contains_locations = True
                results = self.lookup_zipcode(zip_code_country, cleaned_strin
                for i, result in enumerate(results):
                    exec("self.{LC}_list = [result]".format(LC=self.location_

            else:
                places = geograpy.get_geoPlace_context(text=cleaned_string.ca
                for location_context in self.location_contexts:
                    results = eval("places.{LC}".format(LC=location_context))
                    if len(results) > 0:
                        self.contains_locations = True
                        exec("self.{LC}_list.append(places.{LC}[0])".format(L
```

```
        return self.contains_locations, {"countries": self.countries_list, "r
```

In [7]:

```
for example_location in example_locations:
    location_engine = LocationEngine(example_location)
    location_found, locations = location_engine.identify_location()
    print(example_location)
    print("Location found in string = %s" % str(location_found))
    print("Locations found = %s" % str(locations))
    print()
```

```
SLOUGH, SE12 2XY
Location found in string = True
Locations found = {'countries': ['GB', 'United Kingdom'], 'regions': ['Englan
d'], 'places': []}

33 TIMBER YARD, LONDON, L1 8XY
Location found in string = True
Locations found = {'countries': ['GB', 'United Kingdom'], 'regions': ['Englan
d'], 'places': ['LONDON']}

44 CHINA ROAD, KOWLOON, HONG KONG
Location found in string = True
Locations found = {'countries': ['Hong Kong'], 'regions': [], 'places': ['HONG
KONG']}

Scott House, Suite 1, The Concourse, Waterloo Station, London, England, SE1 7L
Y
Location found in string = True
Locations found = {'countries': ['GB', 'United States', 'United Kingdom', 'Aus
tralia', 'United States'], 'regions': ['England'], 'places': ['London', 'Engla
nd']}

LONDON
Location found in string = True
Locations found = {'countries': ['United Kingdom'], 'regions': [], 'places':
['LONDON']}

HONG KONG
Location found in string = True
Locations found = {'countries': [], 'regions': [], 'places': ['HONG KONG']}

ASIA
Location found in string = True
Locations found = {'countries': [], 'regions': [], 'places': ['ASIA']}

Nevereverland
Location found in string = True
Locations found = {'countries': [], 'regions': [], 'places': ['Nevereverlan
d']}

Planet Saturn
Location found in string = False
Locations found = {'countries': [], 'regions': [], 'places': []}

Jurassic Park
Location found in string = True
Locations found = {'countries': [], 'regions': [], 'places': ['Jurassic Par
k']}

Tatooine
Location found in string = False
Locations found = {'countries': [], 'regions': [], 'places': []}

Nutella Sandwich
Location found in string = False
```

```
Locations found = {'countries': [], 'regions': [], 'places': []}
```

As you can see here we now have an engine that will be able to take information from a string and ascertain whether there is a location within it or not. It is not perfect in any strech of the imagination as it misses out major location names such as Hong Kong or Asia

Calculating Physical Goods

For this category, there will have to be something a little different to establish if:

1. The string is referencing an object at all
2. If the physical good is something that is a collective item or something that is a one off

One approach that would work for the examples would be to use the NLTK parts of speech model (POS) to look for words next to each other that are singular nouns. The Spacy entity recognition model doesn't work very well here.

In [8]:

```
example_physical_goods = ["HARDWOOD TABLE", "Frontloaded Washing Machine", "U
```

In [9]:

```
def physical_goods_engine(string):
    """Checks for named pronouns next to each other and returns a list of the
    physical_objects_found = []
    tokens = nltk.word_tokenize(string)
    tokens = [token.lower() for token in tokens]
    # calculate the parts of speech for every word in the string
    tagged = {k:v for k,v in nltk.pos_tag(tokens)}
    for i, word in enumerate(tagged):
        if i > 0:
            # if there are two nouns in a row, there is a high change that th
            if tagged[word] == "NN" and list(tagged.values())[i-1] == "NN":
                physical_objects_found.append(" ".join([list(tagged.keys())[i

    return len(physical_objects_found) > 0, physical_objects_found
```

In [10]:

```
for example_physical_good in example_physical_goods:
    print(physical_goods_engine(example_physical_good))
```

```
(True, ['hardwood table'])
(True, ['washing machine'])
(False, [])
(True, ['plastic bottle'])
(True, ['jesus christ'])
(True, ['bruce pannaman'])
(False, [])
(True, ['recipe book'])
(False, [])
(True, ['kitchen table'])
```

This approach works remarkably well for physical goods with 2 words in them. It does have some errors though:

1. It cannot recognise the difference between names and objects. This coiuld be negated by cross referencing it against a dataset of Person names to rule this out, however, it will not be fool proof e.g. for something like a designer furniture called the "Rebecca Chair"

2. This approach is limited to looking at Bi-grams, where as a physical good can be more or less than 1 word.

If there was a third party API that could be used to verify physical good categories, this would improve performance significantly. I would suggest something like the Amazon products api, or a similar version from Aliexpress or Ebay. I would use the API to search for the term and assess whether there is a physical good based on the number of items available to buy that come back.

Serial Numbers

When identifying serial numbers, I would suggest that the main determination will be whether the string contains non-nouns of which are not in the dictionary.

To do this I will use the NLTK parts of speech model combined with a library called Pyenchant to assess if the word is in the dictionary or not. If a third or more of the tokens in a phrase are not nouns and also not in the dictionary, the string is more than likely fits into the serial number category

In [11]:
```
import enchant
# brew install enchant
dictionary = enchant.Dict("en_GB")
```

In [12]:
```
example_serial_numbers = ["XYZ 13423 / ILD", "ABC/ICL/20891NC", "stc112hJJ5",
```

In [13]:
```
def serial_number_engine(string):
    """Assesses whether the string is a serial number based on NLTK pos model
    tokens = re.split('/|\W',string)
    tokens = [token.lower() for token in tokens]
    tagged = {k:v for k,v in nltk.pos_tag(tokens) if len(k) > 0}
    # If there is only one word, just check if it is in the dictionary using
    if len(tagged) == 1:
        non_dictionary_words_found = [word for word, pos in tagged.items() if
    # For multiple words in the string, firstly check for non-noun words (ind
    else:
        non_dictionary_words_found = [word for word, pos in tagged.items() if
    # If the % of words in the string that arn't in the dictionary are > 1/3,
    return len(non_dictionary_words_found)/len(tokens) >= 0.33, string
```

In [14]:
```
for example_serial_number in example_serial_numbers:
    print()
    print(example_serial_number)
    print(serial_number_engine(example_serial_number))
```

```
XYZ 13423 / ILD
(True, 'XYZ 13423 / ILD')

ABC/ICL/20891NC
(True, 'ABC/ICL/20891NC')

stc112hJJ5
(True, 'stc112hJJ5')

xn256 1jj
(True, 'xn256 1jj')
```

```
Smart Camel
(False, 'Smart Camel')

Lemon Pancake
(False, 'Lemon Pancake')
```

Company names

For company names, the spacy named entity model works nicely again to establish ORG named entities. However, as per the spec we have to deal with:

1. Abbreviations
2. International entities
3. Company structures e.g. inc. ltd. org.

In [15]:

```
company_name_examples = ["Marks and Spencers Ltd", "M&S Limited", "NVIDIA Ire
```

In [16]:

```
def company_engine(string):
    """Used the Spacy Named entity recognition model to assess whether there
    doc = nlp(company_name_example)
    for ent in doc.ents:
        # Can the Named Entity Recognition model find an organisation level e
        if ent.label_ == "ORG":
            return True, ent.text
    return False, string
```

In [17]:

```
for company_name_example in company_name_examples:
    print(company_engine(company_name_example))
```

```
(True, 'Marks')
(True, 'M&S Limited')
(False, 'NVIDIA Ireland')
(True, 'Apple Inc.')
(False, 'Vector.ai')
(False, 'Bruce Pannaman')
(False, 'Henry the Hoover')
```

This approach is not perfect as it has missed an entity like NVIDIA. An extra lookup could be used to confirm this using an third-party company data API such as Intrinio, which uses data from public listings on stock exchanges to return data around the company. I would look for n-grams that have a Proper Noun followed by a geographic marker to identify a company name such as "NVIDIA Ireland" to parse against the API.

This approach would be good to get additional data to supplement the company once we have grouped them together.

Grouping entities together in a sequence

The second part of this test is having the ability to ingest and process strings and not only separate them into the categories above but also group similar entities together. The ways I intend to do this within this task are:

1. Manage conjunctions such as "&", "and", "-" etc.

2. Handle abbreviations e.g. M&S = Marks and Spencers
3. Bring together similar worldwide entities e.g. Amazon Europe Ltd. = Amazon Ltd.

In [18]:

```
class EntityEngine:
    def __init__(self):
        self.entities = {"company_names": [], "company_addresses": [], "seria
        self.company_stop_words = ["ltd", "inc", "org", "gmbh", "&"]
        self.abbreviation_regex = r"\b[A-Z]{2,}\b"

    def categorise_entities(self, input_string):
        """Use each of the engines created above to identify each category an
        location_found, locations = LocationEngine(input_string).identify_loc
        if location_found:
            self.handle_duplicate_entities("locations", locations)

        physical_object_found, physical_good = physical_goods_engine(input_st
        if physical_object_found:
            self.handle_duplicate_entities("physical_goods", physical_good)

        serial_number_found, serial_number = serial_number_engine(input_strin
        if serial_number_found:
            self.handle_duplicate_entities("serial_numbers", serial_number)

        company_name_found, company_name = company_engine(company_name_exampl
        if company_name_found:
            self.handle_duplicate_entities("company_names", company_name)

    def handle_duplicate_entities(self, category, string):
        """Takes each string through the find_similarity() method to try and 
        # Go through each entity of the found category to see if there is a s
        for other_entity in self.entities[category]:
            if self.find_similarity(other_entity, string):
                other_entity += " + %s" % str(string)
                return
        # If no similarity found, add it to the end of the category list
        self.entities[category].append(str(string))

    def clean_string(self, string, lower=True):
        """Removes company and NLTK stop words from the string. Can lowercase
        if lower is True:
            return " ".join([word for word in str(string).lower().split(" ") 
        else:
            return " ".join([word for word in str(string).split(" ") if word 


    def find_similarity(self, previous_string, new_string):
        """Goes through some of the similarity edge cases to see of an entity
        previous_string_clean = self.clean_string(previous_string)
        new_string_clean = self.clean_string(new_string)

        # Find the same word (excluding stop words) in each entity string
        if any([word in new_string_clean.split(" ") for word in previous_stri
            return True

        # Look for abbreviations in both previous strings and next strings. A
        abbreviations_previous_string = re.findall(self.abbreviation_regex, se
        abbreviations_new_string = re.findall(self.abbreviation_regex, self.c

        # abbreviation found in previous string
        if len(abbreviations_previous_string) > 0:
            new_string_words = self.clean_string(new_string, lower=False).spl
            for i in range(len(new_string_words)):
                if i <= len(new_string_words) + len(abbreviations_previous_st
```

```
                    if new_string_words[i: i+len(abbreviations_previous_strin
                        return True

        # abbreviation found in new string
        elif len(abbreviations_new_string) > 0:
            previous_string_words = self.clean_string(previous_string, lower=
            for i in range(len(previous_string_words)):
                if i <= len(previous_string_words) + len(abbreviations_new_st
                    if previous_string_words[i: i+len(abbreviations_new_strin
                        return True

        return False
```

In [19]:

```
example_strings = ["MARKS AND SPENCERS LTD", "LONDON", "ICNAO02312", "LONDON,
```

In [20]:

```
ee = EntityEngine()

for example_string in example_strings:
    ee.categorise_entities(example_string)
```

At this point, I have spent a couple of hours on this and will stop here.

Hopefully from what I have documented here, I have shown my thinking around this problem and highlighted the next steps to make this system better by adding extra use cases and other data sources.