

What Do People Want?

Using Machine Learning to Objectively Identify Our Motivations

By Oliver Morris

oliver.john.morris@gmail.com

9th July 2021

Inspiration

Bertrand Russell proposed 4 fundamental desires of political importance. He did this in his acceptance speech for the Nobel Prize for Literature, 1950:

<https://www.nobelprize.org/prizes/literature/1950/russell/lecture/>

You can listen to him give the speech at:

http://www.kpfahistory.info/dand1/bertrand_russell_%20what_desires_are_politically_important.mp3

Russell asks us to consider two broad sets of desire; satiable and insatiable. Both have an effect on history, although satiable desires tend to have small cycles, e.g. hunt, eat and repeat. Whereas insatiable desires can have cycles which arc over decades, eg the acquisition, exercise and eventual loss of power.

Examples of Satisfiable Desires:

- Physical needs (air, thirst, hunger)
- Safety
- Companionship (sexual or otherwise)

Examples of Insatiable Desires:

- Acquisitiveness ("the wish to possess as much as possible")
- Rivalry ("a much stronger motive")
- Vanity ("a motive of immense potency")
- Love of Power ("which outweighs them all")

He admits that these 'insatiable' desires appear ethically neutral at best. He does not deny that more altruistic motivations exist, such as compassion, but that they are less powerful.

What would Machine Learning tools find if asked to trawl Wikipedia for its millions of references to wishes, wants and desires? In its descriptions of lives and history Wikipedia promises fertile territory for Russel's 'insatiable' desires. This is in contrast to data sources such as tweets and facebook, which may over represent the satiable desires.

We do not expect an answer more intelligent than that offered by the Nobel prize winning philosopher. However, it would be fascinating to see what objective mathematical tools cause to fall out of the data, without the preconceptions of a human being.

Not a Simple Clustering Problem

As a data scientist it is worth having some background knowledge about how psychology and philosophy professionals approach desire. Wikipedia has an excellent introduction at <https://en.wikipedia.org/wiki/Desire>

Reading the article quickly highlights some analogies with data science. The basic classification of desire into 'Action', 'Pleasure' and 'Value' is reminiscent of Reinforcement Learning with its Action, Value and Quality (Q learning) approach.

It is also interesting to note that psychologists may not classify desire as an emotion. Emotions are often presented on Plutchik's wheel of emotion, which is then elaborated upon with Plutchik's dyads. The dyads illustrate how basic emotions are mixed to form more complex emotions, like a colour wheel. Yet, desire is not amongst any of those basic emotions or their various combinations, instead desire has been called a 'ur-emotion'. Frijda and Parrott in 2011 wrote "...ur-emotions [are] the functionally central underlying processes of action readiness, which are not emotions at all. We propose that what is basic and universal in emotions are ...states of action readiness, themselves variants of motive states to relate or not relate with the world and with oneself. Unlike emotions, ur-emotions can be held to be universal and biologically based."

That statement needs to be unpacked and much of the Wikipedia entry on desire is concerned with the various classifications of desire which aid either psychological or philosophical analysis. It is important to note that the investigation detailed in this paper will not test for any of these classes of desires, no preconceptions of classification will be presented to the algorithm at all. The objective is for the data to speak for itself. However, the analysis will be concerned with how a classification algorithm can be tuned to find themes of desire, not topics of desire. Topic modelling is a well developed subject with many tools available to us, but consider how such an algorithm would group (i.e. cluster) the following statements:

- I want to become a professional football player
- I want to be qualified as an aircraft pilot
- I want to keep the football for myself
- I want to fly the aircraft solo

A topic modelling tool, particularly if base don word frequency, would almost certainly find two clusters; football and aircraft. What are the chances that the tool would instead find clusters for ambition and possessiveness? The clues are there but the signal from the topics is surely stronger.

Peter Turney, an NLP researcher, writes, "Relational similarity is correspondence between relations, in contrast with attributional similarity". Relational similarity, which Mr Turney also calls 'semantic similarity', is the problem faced in this analysis.

Summary of Results

The results of this analysis are discussed at the end of this paper. In summary, the following clusters are found for sentences and clauses in Wikipedia containing the words 'want', 'wanted' or 'wants'. This represents the automated clustering of over 180,000 such sentences. The algorithm confidently assigned 73% i.e. 130,000+ of those sentences to a cluster. It was unable to confidently place the remainder in any single cluster. The clusters were independently found by the clustering algorithm, it was not told how many clusters to seek.

- Desire for emotional connection
- Desire for emotional distance
- Desire for affiliation with a group
- Desire for an end to affiliation
- Wishes negotiated between parties
- Obstinacy in relationships
- Desires about other's perspectives
- Desire to win authority via allies
- Desire to impose authority over territory
- Desire to impose authority over others
- Desire for control
- Desire for independence

The above descriptions of each cluster are, of course, just summaries formed from small selections of sentences at the heart of each cluster. If we could travel across a cluster and take samples at regular intervals then possibly we would observe these central themes morphing into something more generic and then into the theme of a neighbouring cluster. Alternatively, we might observe very little and find it difficult to follow the algorithm's clustering the further we get from the densest part of each cluster. This is work yet to be done.

We may have expected to find variations on Russell's proposed desires; ambition, acquisitiveness, vanity etc and to some extent we can see those in the above clusters. However, we also see a unifying theme to these clusters. They are all desires defined by our relationships with each other. They are not driven by objects or objectives, other than the objective of being relevant to each other. Some clusters focus on our relationships in large groups (teams, governments), others focus on our relationships in small groups (friends, family), but all are motivations defined by ourselves within a group.

This is a fascinating and even rather beautiful outcome, albeit constrained by the limits of this anlaysis, which are discussed at the end of the paper.

Approaches

The first step will be acquire the data, which is a cleaned up version of the entirety of wikipedia from a snapshot made in 2014. This is a standard dataset available form Huggingface and used in many data science models. Using wikipedia means we will get a narrative of history, which is our and Russell's interest, as opposed to personal concerns which would be found in social media. The 'BookCorpus' of literature, also available from Huggingface, is an interesting alternative but is not investigated here.

We will then use regex to process the wikipedia entries, selecting clauses or sentences containing one of these keywords; 'want', 'wish' or 'desire' or their equivalents for other tenses/person (wanted, wished, etc). We anticipate hundreds of thousands of instances of such text in Wikipedia. Having acquired all instances of clauses or sentences including those words, how should this data be analysed?

Approach 1

Transformers are the most powerful NLP tool currently available. OpenAI's famous GPT-3 is based on a Transformer. They can be used to convert text into a numeric format which can then be analysed with statistical tools. This numeric format is simply a vector per word, called an 'embedding'.

Google's BERT, originally released in 2018, has become a successful and standard approach for many NLP applications. ALBERT is a simplified version of BERT, easier to run on a desktop PC, but still highly effective. All BERT and ALBERT word embeddings are contextual, meaning the vector assigned to each word in each sentence is unique and depends on its context. So, the embedding for 'want' is unique in each sentence and will reflect the context in which it is used. This 'contextual' aspect may be useful when seeking relational similarities, so this is our first approach:

- Acquire sentence embeddings
 - Taken from a standard transformer, such as Huggingface's ALBERT, and then averaged across the word tokens in each sentence to give us an embedding for each sentence
 - Or use the sentence embeddings given by a tool dedicated to creating such embeddings, such as SBERT
- Compress those embeddings down to a manageable number of dimensions
 - Using either PCA or an autoencoder
- Cluster the data
 - Using tools such as HDBSCAN where the number of clusters is allowed to set by the data itself, not imposed as it would be with KNN or GMM.
- Analyse the clusters
 - By visual inspection to see what similarities sentences in each cluster share
 - By using topic modelling tools such as TF-IDF, to identify whether clusters have formed around topics or relational similarities

If this approach yields convincing results, then investigation can cease. However, the concern is that averaging transformer embeddings for each token in a sentence will encourage clusters to form around similarity of attributes, rather than similarity of relationships, ie topic clustering not semantic clustering. For example, clusters would form around desires with respect to music, politics or family. Whereas we seek desires such as ambition, which are agnostic to the setting in which they are found. TF-IDF is a word frequency measure, so can be used to characterise clusters by how dominated they are by a given attribute or topic.

BERT transformer embeddings contain a lot of information and have 768 dimensions. Therefore, the code in this approach is dominated by the need for appropriate data compression.

Approach 2

This approach is informed by the paper "Sentence Analogies: Linguistic Regularities in Sentence Embeddings", Zhu & deMelo 2020 [<https://aclanthology.org/2020.coling-main.300.pdf>]. They recommend using GloVe embeddings and a function called 3CosMul to measure the 'analogy' distance between those embeddings. Both those embeddings and that measure date from 2014, so are old by NLP standards, but apparently work better than BERT in this problem. Zhu & deMelo did not attempt clustering using this combination, however, deMelo publishes an excellent list of related NLP resources for similar projects [<http://gerard.demelo.org/projects.html>].

GloVe embeddings have only 50 dimensions. So this approach is much easier than Approach 1, requiring very little work on data compression.

The approach adopted in this paper is as follows:

- Convert words in each sentence to GloVe embeddings
- Identify specific embedding for the key searchword in each sentence; 'want', 'desire', 'wish' etc
 - uses Gensim package
- Derive sentence embeddings from word embeddings
 - Either using Discrete Cosine Transforms (DCT), or simple averaging of word embeddings
- Calculate 3CosMul score for each analogy pair, hence pairwise distance matrix
 - analogy pair = searchword A in sentence A vs searchword B in sentence B
 - eg score to what extent 'want' in 'I want food' is analogous to 'wish' in 'I wish to eat'
- Reduce dimensionality of distance matrix to 30 dims or less
 - possibly using PCA since GloVe has standard embeddings with only 50 dimensions, reduction to 30 is expected to be straightforward
- Use HDBSCAN, which permits use of a custom distance measure such as 3CosMul, to cluster the pairwise distances
 - explore the effect of clustering parameters such as min_cluster_size.
- Analyse clusters by inspection and TF-IDF
 - If the model is truly attribute agnostic, then TF-IDF should struggle to find topics based on attributes
 - compare with clusters discovered using the first approach

Acquiring the Data

The following code chunks will be concerned with downloading the Wikipedia data, Wiki40b, and extracting the clauses required for this investigation. This data can be found at:

- <https://huggingface.co/datasets/wiki40b>

As an aside, it should be noted that there is an 'analogy' dataset already available from Google:

- [https://aclweb.org/aclwiki/Google_analogy_test_set_\(State_of_the_art\)](https://aclweb.org/aclwiki/Google_analogy_test_set_(State_of_the_art))
- <http://download.tensorflow.org/data/questions-words.txt>

However, this is a word analogy dataset, not a sentence analogy dataset. Furthermore, it is focussed on syntactic analogies, not the semantic analogies we seek. For example:

- London is to England as Rome is to Italy
- Cheerful is to cheerfully as free is to freely

This analogy dataset has been reviewed but is not of use to this project. Interestingly, if this project is successful in finding relational similarities within sentences then it may usefully provide a new set of training set for analogies for relational similarities (i.e. analogies).

In [1]:

```
# file management
import os

# the huggingface datasets
from datasets import load_dataset

# data wrangling
import re
import pandas as pd
import numpy as np
from ast import literal_eval
from itertools import product

# charting
from matplotlib import pyplot
import plotnine as p9
import pylab

# process timing
import tqdm.notebook as tq
import time
import datetime

# statistical learning
import random
import math
import umap
import hdbscan
import scipy.stats as ss
import statistics as st
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from scipy.spatial import distance

# AzureML basics
import azureml.core
from azureml.core import Workspace, Experiment, Run, Environment, ScriptRunConfig, Dataset

# AzureML compute
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.exceptions import ComputeTargetException

# AzureML Hyperdrive tools
from azureml.train.hyperdrive import GridParameterSampling, choice, HyperDriveConfig, PrimaryMetricGoal, BanditPolicy
from azureml.widgets import RunDetails

# AzureML environment tools
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core.runconfig import RunConfiguration

# Pretrained Pytorch NLP Model from Huggingface
import torch
from transformers import AlbertTokenizer, AlbertModel

# deep Learning for Autoencoder
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.layers import Dense, Input, Layer
from tensorflow.keras.callbacks import EarlyStopping

print("Tensorflow Version:", tf.__version__)
print("Is tensorflow executing eagerly?", tf.executing_eagerly())
print("Is tensorflow using GPU?", tf.test.is_built_with_gpu_support())
print("Is tensorflow using Cuda?", tf.test.is_built_with_cuda())

# set working directory
os.chdir('/home/oliver/Documents/Deep/Data')
path = os.getcwd()
```

```

print("Working path is...")
print(path)

Tensorflow Version: 2.4.1
Is tensorflow executing eagerly? True
Is tensorflow using GPU? True
Is tensorflow using Cuda? True
Working path is...
/home/oliver/Documents/Deep/Data

```

In [2]:

```

# get the Wikipedia 2014 dataset, downloaded from Huggingface using the huggingface function load_dataset()
dataset = load_dataset('wiki40b', 'en') #, split='test'

print(dataset.keys())
print(type(dataset['train']))

print('Length of training set :', len(dataset['train']))
print('Length of validation set:', len(dataset['validation']))
print('Length of testing set :', len(dataset['test']))

```

```

Reusing dataset wiki40b (/home/oliver/.cache/huggingface/datasets/wiki40b/en/1.1.0/063778187363ffb294896eaa010fc254b42b73e3111
7c71573a953b0b0bf010f)
dict_keys(['train', 'validation', 'test'])
<class 'datasets.arrow_dataset.Dataset'>
Length of training set : 2926536
Length of validation set: 163597
Length of testing set : 162274

```

View data example

In [3]:

```

# Let's inspect some examples...

# get 100 random samples of the data
sample_list= random.sample(range(len(dataset['validation'])), 100)

small_dataset = dataset['validation'].select(sample_list)

# print the first example
small_dataset[0]

```

Out[3]:

```

{'wikidata_id': 'Q28451460',
 'text': '\n_START_ARTICLE_\nInvisible (Leeland album)\n_START_SECTION_\nAccolades\n_START_PARAGRAPH_\nThe song, "Lion and the Lamb", was No. 8 on the Worship Leader's Top 20 Songs of 2016 list._NEWLINE_On August 9, 2017, the Gospel Music Association w as announced that the song "Lion and the Lamb" would be nominated for a GMA Dove Award in the Song of the Year and Worship Son g of the Year categories at the 48th Annual GMA Dove Awards.',
 'version_id': '3367906420288520607'}

```

Extract Wishes, Wants and Desires

At this stage we're not sure if we want to embed entire sentences with these words, or just the relevant clauses of the sentences It may be that sentences contain so much meaning that they drown out the want/desire, so clauses are better. Alternatively, clauses may give too little context. So, we will extract two data sets. One of the sentences, the other of clauses.

In [4]:

```

# function to extract sentences which have the key words 'wish', 'want', 'desire' and their plurals/tenses

def get_sentences_with_word(the_dataset, searchwords):

    # create empty list to hold our results
    matches = []

    # join keywords into an 'or' statement for regex
    searchwords_joined = '\b(' + '|'.join(searchwords) + ')\b'

    # pattern to extract SENTENCES which contain one of the searchwords
    # Clauses are just sentences split by ','
    # This pattern was built by trial and error in https://regexr.com/
    pattern_sentence = re.compile(r"[A-Z]{1}[^_.!?]*\s+[^_.!?\n]*", re.M)
    pattern_searchwords = re.compile(searchwords_joined, re.IGNORECASE)

    # Loop thru each text
    for i in tq.tqdm(range(len(the_dataset))):

        # get next record
        text      = the_dataset[i]['text']
        wikidata_id = the_dataset[i]['wikidata_id']

        # remove periods (full stops) which aren't sentence endings
        # this is tough, won't be perfect
        text = text.replace("e.g.", "eg")
        text = text.replace("i.e.", "ie")

        # replace any instance of the ASCII encodable separator we use for the pandas file,
        # which is a tilde ~ (ASCII 126)

```

```

text = text.replace("~", "-")

# remove any instance of the wiki40b tokens for end of sentence, replace with the fullstop marker ". "
text = text.replace("\n_START_SECTION_\n", ". ")
text = text.replace("\n_START_PARAGRAPH_\n", ". ")

# extract sentences
try:

    sentences = pattern_sentence.findall(text)

except AttributeError:
    sentences = []

# cycle through sentences looking for matching searchwords
for sentence_ref in range(len(sentences)):

    sentence = sentences[sentence_ref]

    # find any matching words in the sentence
    searchword_matches = pattern_searchwords.findall(sentence)

    # if there are matching words and there more than 4 words in the sentence, grab it and analyse clauses
    if len(searchword_matches) > 0 and len(sentence.split())>4:

        # split the sentence into clauses
        clauses = sentence.split(',')

        # if there is only one clause, just save it and move on
        if len(clauses) == 1:
            # append the matching sentence to the sentence list
            matches.append({'wikidata_id':wikidata_id, 'sentence_ref':sentence_ref,
                           'clause_ref':-1, 'text':sentence})

        # if there are multiple clauses then try to save separately, if each is long enough and contains keyword
        else:
            # reset clause counter
            clause_counter = 0

            # search for the searchwords in each clause
            for clause_ref in range(len(clauses)):

                # strip white space
                clause = clauses[clause_ref].strip()

                # split the clause by word
                clause_words = clause.split()

                # if the clause is more than 5 words AND contains a keyword, then save it
                any_searchword_inclause = any([searchword in clause_words for searchword in searchwords])
                if ( len(clause_words)>5 ) and ( any_searchword_inclause ):

                    # append the matching clause to the clause list
                    matches.append({'wikidata_id':wikidata_id, 'sentence_ref':sentence_ref ,
                                   'clause_ref':clause_ref, 'text':clause})

                # increment clause counter
                clause_counter += 1

            # if we analysed each clause and none qualified on its own, then we save the whole sentence
            if clause_counter == 0:

                matches.append({'wikidata_id':wikidata_id, 'sentence_ref':sentence_ref,
                               'clause_ref':-1, 'text':sentence})

# convert to pandas
matches = pd.DataFrame(matches)

# remove non ASCII characters, because these cannot be encoded by BERT
matches['text'] = matches['text'].apply(lambda x: x.encode('ascii', 'ignore').decode('ascii'))

return matches

```

In [5]:

```

# test the function on the short sample
#small_dataset
searchwords= ['want', 'wants', 'wanted', 'wish', 'wishes', 'wished', 'desire', 'desires', 'desired']
sample_ext = get_sentences_with_word(the_dataset=small_dataset, searchwords=searchwords)
sample_ext.to_csv(path_or_buf=os.path.join(path,'sentences_sample.csv'), index=False, sep='~')

# Its important that we work from the version saved to file because thats what the final version will do
# get first wikidata id
if len(sample_ext)>0:
    sample_ext_id1 = sample_ext.iloc[0,]['wikidata_id']
    print('\n')

```

```

print('Source Data from Wiki40b')
print('\n')
print( sample_ext_id1 )

print('\n')
print('First Extracted Sentence')
print(sample_ext.loc[0,'text'])

#print(pd.read_csv(os.path.join(path, 'sentences_sample.csv'), sep='~'))

```

Source Data from Wiki40b

Q58860

First Extracted Sentence

though for pragmatic reasons: he wanted to help defend middle- and upper-class interests from further disruption

In []:

```

datasplits = ['train', 'test', 'validation']
searchwords = ['want','wants','wanted','wish','wishes','wished','desire','desires','desired']

for split in datasplits:
    the_dataset = dataset[split]
    text_pandas = get_sentences_with_word(the_dataset, searchwords=searchwords)
    # save to file
    text_pandas.to_csv(path_or_buf=os.path.join(path,'sentences_'+split+'.csv'), index=False, sep='~', encoding='ascii')

```

In [9]:

```
# Let's look at some results from memory
text_pandas.sample(n=3)
```

Out[9]:

	wikidata_id	sentence_ref	clause_ref	text
311439	Q5531013	226	-1	Yoon believes Abenomics represents a desire to...
81060	Q2024146	39	-1	And even if she still loves him she has now a ...
237715	Q3133488	33	1	Murerwa said that he did not want to make any ...

In [4]:

```

# Let's also inspect the data we saved to file in the previous chunk
# Keen to prove there are no file encoding problems

sentences_train = pd.read_csv(os.path.join(path,'sentences_train.csv'), sep='~', encoding='ascii' )
sentences_testi = pd.read_csv(os.path.join(path,'sentences_testi.csv'), sep='~', encoding='ascii' )
sentences_valid = pd.read_csv(os.path.join(path,'sentences_valid.csv'), sep='~', encoding='ascii' )

sentences_train.name = 'train'
sentences_testi.name = 'testi'
sentences_valid.name = 'valid'

print("Training Records : ", len(sentences_train))
print("Testing Records : ", len(sentences_testi))
print("Validation Records: ", len(sentences_valid))

```

Training Records : 434481
Testing Records : 24479
Validation Records: 25014

In [117...]

```
# inspect a sample of the data so far
dataset.sample(n=3)
```

	wikidata_id	sentence_ref	text
549	Q45810	54	the King gave his opening speech to his first English Parliament in which he spoke of his desire to secure peace
22191	Q7717884	331	as they were the last of the original teams still standing and he wanted her to be able to stay
8808	Q465646	37	Peirce wanted to tell the story from Brandon's perspective

Using ALBERT to get Embeddings from Text

We need a way to 'embed' these clauses and sentences, ie convert them to a meaningful, multidimensional, vector. The approach will be to pass each clause or sentence into the ALBERT transformer model. We'll then take the hidden states of the model with that sentence as our 'embedding'. This is a common approach and reliable since it uses the pre-trained ALBERT model, which is 'state of the art' in a number of tasks, not least sentence sentiment classification, a closely related task.

Albert is a 'A Lite' version of the BERT model. Albert was first published in September 2019 and then presented at the ICLR conference in April 2020, 11 months before this notebook was written.

We'll use Huggingface's PyTorch based ALBERT transformer and associated tokenizer to process the text into embedding vectors which we can then use for clustering. <https://huggingface.co/albert-base-v2>

This transformer model has the following configuration:

```
12 repeating layers  
128 embedding dimension  
768 hidden dimension  
12 attention heads  
11M parameters
```

ALBERT is particular in that it shares its layers across its Transformer. Therefore, all layers have the same weights. Using repeating layers results in a small memory footprint, however, the computational cost remains similar to a BERT-like architecture with the same number of hidden layers as it has to iterate through the same number of (repeating) layers

```
In [5]: # AlbertModel = The bare Albert Model transformer outputting raw hidden-states without any specific head on top.  
  
# Load pre-trained model tokenizer (vocabulary)  
# this is the Huggingface AlbertTokenizer for the AlbertModel:  
# https://huggingface.co/transformers/model_doc/albert.html#albertmodel  
# the tokenizer has a vocabulary of 30,522 tokens:  
  
tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
```

Encoding Text in Preparation for Tokenisation

Albert can take as input either one or two sentences, and uses the special token SEP to differentiate them. The CLS token always appears at the start of the text, and is specific to classification tasks. The tokenizer ensures these special tokens are present. Note, the CLS and SEP tokens output embeddings of their own, which are used by some models, such as Sbert, to provide embeddings for the entire sentence. We will not be taking this approach, instead using the embeddings of each token in the sentence.

Both SEP and CLS tokens are always required for all inputs. That's how Albert was pre-trained, and so that's what Albert expects to see.

- 2 Sentence Input:
 - [[CLS]] The man went to the store. [[SEP]] He bought a gallon of milk.
- 1 Sentence Input:
 - [[CLS]] The man went to the store. [[SEP]]

We must encode our sentences and clauses into the format Albert expects. The following code chunk gives an example of how this is done using Huggingface:

```
In [6]: # here's an example  
clause = "Here is the sentence I want embeddings for."  
marked_text = "[CLS] " + clause + " [SEP]"  
  
# Tokenize our sentence with the Albert tokenizer.  
tokenized_text = tokenizer.tokenize(marked_text)  
  
# Print out the tokens.  
print(tokenized_text)
```

['[CLS]', '_here', '_is', '_the', '_sentence', '_i', '_want', '_emb', 'ed', 'ding', 's', '_for', '.', '[SEP]']

The Tokenizer's Output

Notice how the word "embeddings" is represented:

```
'_emb', 'ed', 'ding', 's'
```

The original word has been split into smaller subwords and characters. The underscores preceding some of these subwords are just our tokenizer's way to denote that this subword or character is prefixed with a space. This is because Albert uses a 'sentencepiece' approach to tokenisation whereas the original BERT used a 'WordPiece'. Sentencepiece simply breaks words into their subwords, which is important in languages like German which tend to roll words together without spaces. The Tokenizer then uses the unigram approach to further tokenise the words. See this link for an explanation:

https://huggingface.co/transformers/tokenizer_summary.html#unigram

The unigram Tokenizer deals with situations where a subword token is must do distinct from an identical complete word token, eg 'embedding' may have subword 'bed', which must be distince from 'bed' for sleeping.

The default vocabulary limit of the Albert tokenizer model is 30,000 tokens. See above link for how this vocabulary is built.

In [7]:

```
def get_indexed_tokens(text):

    # Add the special tokens.
    marked_text = '[CLS] ' + text + ' [SEP]'

    # Split the sentence into tokens.
    try:
        tokenized_text = tokenizer.tokenize(marked_text)
    except:
        tokenized_text = ['[CLS]', '[SEP]']

    # sometimes more than 512 tokens result, the model exceeds its limits
    # This throws an error which we handle by simply returning the start and end tokens
    # Later these clauses will be removed from the pandas tables of train/test/validate data
    if len(tokenized_text) > 512:
        tokenized_text = ['[CLS]', '[SEP]']

    # Map the token strings to their vocabulary indices.
    indexes_of_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)

    return indexes_of_tokens
```

In [8]:

```
# from concurrent.futures import ThreadPoolExecutor, as_completed
# Above commented line is for parallel execution, but not used due problems with inadvertent mixing of indexes

# tokenise sentences (inc clauses) for all datasets; training, testing and validation.
for filename in ['sentences_train.csv', 'sentences_testi.csv', 'sentences_valid.csv']:

    # monitor progress
    start_time = time.time()
    print('Opening File : ', filename)

    # Load file of saved clauses
    sentences_ds = pd.read_csv(os.path.join(path, filename), sep='~')

    # this line to ensure no extraneous columns, can happen if re-using this code on same folder
    sentences_ds = sentences_ds[['wikidata_id', 'sentence_ref', 'clause_ref', 'text']]

    # tell us what's happening!
    print('Processing: ', filename, '\n')

    # get the indexes of the tokens
    indexed_tokens = [get_indexed_tokens(text) for text in tq.tqdm(sentences_ds['text'])]

    # record full results in new column in pandas dataframe
    sentences_ds['text_idxs'] = indexed_tokens

    # save pandas dataframe to file
    sentences_ds.to_csv(path_or_buf=os.path.join(path, filename), index=False, sep='~')

    print('Time to process ', filename, ': ', (time.time() - start_time)/60, 'mins')
```

Opening File : sentences_train.csv
Processing: sentences_train.csv

Time to process sentences_train.csv : 1.2162874380747477 mins
Opening File : sentences_testi.csv
Processing: sentences_testi.csv

Time to process sentences_testi.csv : 0.07541091442108154 mins
Opening File : sentences_valid.csv
Processing: sentences_valid.csv

Time to process sentences_valid.csv : 0.07030308246612549 mins

In [9]:

```
# Let's look at what we made...
sentences_ds.sample(n=3)
```

Out[9]:

	wikidata_id	sentence_ref	clause_ref	text	text_idxs
4204	Q959120	10	-1	From the 1950s on, he appeared as a guest star...	[2, 37, 14, 1715, 18, 27, 15, 24, 682, 28, 21,...
2550	Q458749	113	1	Silva said: "I want to either fight a catchwei...	[2, 10559, 87, 45, 13, 7, 49, 259, 20, 694, 10...
15496	Q31204331	8	-1	Spencer is intrigued and decides that he want...	[2, 6212, 25, 21415, 17, 6674, 30, 24, 2846, 2...

```
In [10]: def get_pandasfromcsv(filename, list_cols):
    data = pd.read_csv(os.path.join(path,filename), sep='~')
    # The Pandas to_csv() function stores lists as strings, eg [1,2] becomes '[1,2]', 
    # this is because List objects are not native to the cells of pandas dataframes
    # we could have pickled the pandas object, but easier to simply 'evaluate' the string literal into a list.

    for col in list_cols:
        data[col] = [literal_eval(cell) for cell in data[col]]

    return data
```

```
In [11]: sentences_train = get_pandasfromcsv('sentences_train.csv', ['text_idxs'])
sentences_testi = get_pandasfromcsv('sentences_testi.csv', ['text_idxs'])
sentences_valid = get_pandasfromcsv('sentences_valid.csv', ['text_idxs'])

sentences_train.name = 'train'
sentences_testi.name = 'testi'
sentences_valid.name = 'valid'

print('How many records do we have?')
print(sentences_train.name, ' : ', len(sentences_train))
print(sentences_testi.name, ' : ', len(sentences_testi))
print(sentences_valid.name, ' : ', len(sentences_valid))

How many records do we have?
train : 434481
testi : 24479
valid : 25014
```

Contextualised Embeddings

The BERT transformer and its descendent, Albert, convert tokens to 'contextualised embeddings'. This means it will give a different vector for the same word depending on its context. Of course, contexts are nearly always unique in sentences, so the embeddings for a given word are nearly always unique. Sometimes the word will have a very different meaning in different contexts. Consider the word 'bears' in these contexts:

- He bears arms
- He arms bears

Same word different meaning. This is the power of BERT, whereas Glove or Word2Vec would have given the same embedding for each context. BERT goes further, it will give a slightly different embedding even where the word has the same meaning, but the context is slightly different. For example:

- The doctors by the cars are ugly
- The cars by the doctors are ugly

BERT gives different embeddings for the word 'doctors' in these contexts because the subject/object are different. The above example is given in a paper from Johns Hopkins, a university with a lot of doctors. The paper investigates BERT's embeddings and how they change according to context: <https://www.aclweb.org/anthology/2020.coling-main.325.pdf>

It will prove useful to capture the embeddings for the searchword in each clause. For that, we need to identify the token(s) for each searchword:

```
In [12]: # find the tokens for the searchwords
# which tokens represent the chosen keywords
searchwords= ['want', 'wants', 'wanted', 'wish', 'wishes', 'wished', 'desire', 'desires', 'desired']

# get the embeddings by submitting the tokens to the BERT model, returns np.array
searchwords_tokens = []

for word in searchwords:

    tokenized_word = tokenizer.tokenize(word)
    indexes_of_tokens = tokenizer.convert_tokens_to_ids(tokenized_word)
    searchwords_tokens.append(indexes_of_tokens[0])

searchwords_tkn = pd.DataFrame({'searchword':searchwords, 'searchword_tkn':searchwords_tokens})

# view results
display(searchwords_tkn)
```

	searchword	searchword_tkn
0	want	259
1	wants	2846
2	wanted	417

	searchword	searchword_tkn
3	wish	2536
4	wishes	8803
5	wished	5098
6	desire	3150
7	desires	18374
8	desired	9264

We need to confirm we can find at least one searchword's tokens within the tokenized texts. The below code shows there are examples where we can't find any searchword's token. This is where the searchword is conjugated with a non-alphanumeric character, such as ", , [or -.

Sentences afflicted in this way need to be removed from the data.

```
In [ ]: # find the searchword tokens within the tokenized texts...
text_idxs_match_arg = [[text_idxs.index(searchtoken) for searchtoken in searchwords_tkn['searchword_tkn']]
                      if searchtoken in text_idxs]
                     for text_idxs in sentences_train['text_idxs']]]

# identify where we CANNOT find any searchword tokens
no_match_mask = [len(item)==0 for item in text_idxs_match_arg]
no_match_texts = sentences_train.loc[no_match_mask, 'text'].copy().reset_index(drop=True)
no_match_texts = pd.DataFrame(no_match_texts)

# display some examples of non-matches
pd.set_option('display.width', 2000)
pd.set_option('display.max_colwidth', 2000)
display(no_match_texts.sample(n=6))
```

```
In [14]: # create a lookup (i.e. pandas Series) such that we can enter a token and be returned a searchword
searchwords_tkn_lookup = searchwords_tkn.set_index('searchword_tkn')
searchwords_tkn_lookup = searchwords_tkn_lookup['searchword']

# for each dataset we will find and then remove the afflicted sentences. Finally re-save the data
for dataset in [sentences_train, sentences_testi, sentences_valid]:

    # find the searchword tokens within the tokenized texts...
    text_idxs_match_arg = [[text_idxs.index(searchtoken)
                           for searchtoken in searchwords_tkn['searchword_tkn']]
                           if searchtoken in text_idxs]
                           for text_idxs in dataset['text_idxs']]]

    # save these token locations to the data
    dataset['searchword_tkn_idx'] = text_idxs_match_arg

    # identify where we CAN find at least one searchword's token, i.e. where we have more than zero tokens
    match_mask = [len(item)!=0 for item in text_idxs_match_arg]

    # filter the data accordingly, removing afflicted rows
    dataset_new = dataset[match_mask]

    # for reference, which searchwords were found? Now we use that lookup
    dataset_new['searchwords'] = [[searchwords_tkn_lookup[tokens[idx]] for idx in tkn_idx]
                                 for tkn_idx, tokens in
                                 zip(dataset_new['searchword_tkn_idx'], dataset_new['text_idxs'])]

    # save to csv
    dataset_new.to_csv(path_or_buf=os.path.join(path, 'sentences_{}{}.csv'.format(dataset.name, dataset.name)),
                        index=False, sep='~', encoding='ascii')
```

```
In [15]: # Load the data, now without the afflicted rows

sentences_train = get_pandasfromcsv('sentences_train.csv', ['text_idxs', 'searchword_tkn_idx'])
sentences_testi = get_pandasfromcsv('sentences_testi.csv', ['text_idxs', 'searchword_tkn_idx'])
sentences_valid = get_pandasfromcsv('sentences_valid.csv', ['text_idxs', 'searchword_tkn_idx'])

sentences_train.name = 'train'
sentences_testi.name = 'testi'
sentences_valid.name = 'valid'

print('How many records do we have?')
print(sentences_train.name, ': ', len(sentences_train))
print(sentences_testi.name, ': ', len(sentences_testi))
print(sentences_valid.name, ': ', len(sentences_valid))
```

How many records do we have?
train : 434481

```
testi : 24479
valid : 25014
```

In [16]:

```
# inspect the data
display(sentences_train.sample(n=4))
```

	wikidata_id	sentence_ref	clause_ref	text	text_idxs	searchword_tkn_idx	searchwords
338737	Q5255025	28	-1	It was reported that Rosenblatt wanted a price closer to \$3 billion for Demand Media	[2, 32, 23, 1117, 30, 10735, 22360, 417, 21, 2162, 1788, 20, 5341, 2786, 26, 3888, 941, 3]	[7]	['wanted']
90104	Q32751865	132	-1	His last wish for his five grandchildren was to urge them to continue family tradition	[2, 33, 236, 2536, 26, 33, 355, 16739, 23, 20, 7773, 105, 20, 1816, 190, 2876, 3]	[3]	['wish']
151619	Q519232	17	1	Juvencus announces that he wishes to meet the lying tales of the pagan poets	[2, 3191, 3124, 4636, 20813, 30, 24, 8803, 20, 1255, 14, 3262, 6495, 16, 14, 12094, 10537, 3]	[7]	['wishes']
203849	Q226773	35	3	"They want this one to do well so they can go ahead with the sequel	[2, 13, 7, 2077, 259, 48, 53, 20, 107, 134, 86, 59, 92, 162, 2173, 29, 14, 6998, 3]	[4]	['want']

Clause Length

Some clauses are very long and some very short. All clauses will need to be padded to the same length for submitting to the model. So if we simply pad all clauses to the longest possible, then we will have huge tensors of mostly zeros (padding). Also, some clauses are very short and contain effectively no information from which the model can learn. It will be better to crop the unusually long and unusually short clauses.

To decide what max length to use we'll need to see the distribution of lengths, then aim for something like the 95th percentile.

In [17]:

```
token_lengths = {}

for dataset in [sentences_testi, sentences_valid, sentences_train]:
    token_lengths[dataset.name] = [len(token_idxs) for token_idxs in dataset['text_idxs']]

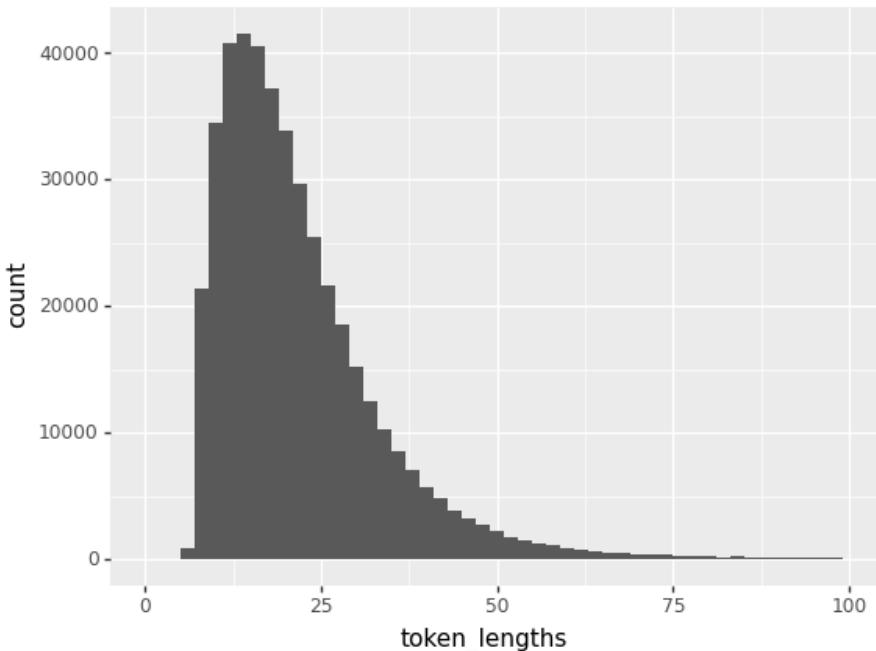
# where are the percentiles?
print('The longest text has a token length of: ', np.max(token_lengths['train']))
print('The percentiles of text lengths are [10th, 30th, 50th, 70th, 90th] ', np.percentile(token_lengths['train'], [10,30,50,70,90]))

print('\n')

# chart the distribution of token lengths as a histogram
hist_data = pd.DataFrame({'token_lengths':token_lengths['train']})
(
    p9.ggplot(hist_data, p9.aes(x='token_lengths'))
    + p9.geom_histogram(binwidth=2)
    + p9.scales.xlim(0,100)
)
```

```
The longest text has a token length of: 506
The percentiles of text lengths are [10th, 30th, 50th, 70th, 90th]  [11. 15. 20. 25. 37.]
```

```
/home/oliver/anaconda3/envs/deep_gpu/lib/python3.8/site-packages/plotnine/layer.py:372: PlotnineWarning: stat_bin : Removed 11
20 rows containing non-finite values.
/home/oliver/anaconda3/envs/deep_gpu/lib/python3.8/site-packages/plotnine/layer.py:467: PlotnineWarning: geom_histogram : Remo
ved 2 rows containing missing values.
```



Out[17]: <ggplot: (8792551226576)>

In [18]:

```
# the charting package is warning about missing and non-finite values.
# Let's find any such instances in the data. If none, then assume the processing has caused the errors, not the data

print('Missing data:')
print(hist_data['token_lengths'][hist_data['token_lengths'].isna()])
print('\n')
print('Non finite data:')
print(hist_data['token_lengths'][np.isinf(hist_data['token_lengths'])])
```

Missing data:
Series([], Name: token_lengths, dtype: int64)

Non finite data:
Series([], Name: token_lengths, dtype: int64)

All sentences will have to be padded to the length of the longest sentence in the dataset. This will make for a huge amount of data, of which the vast majority will just be '0', the padding token. It will also increase the number of weights in the model and hence the training time. To simplify the project and work within our resources, we need to remove the longer clauses.

Furthermore, clause embeddings will be extracted by averaging over all the token embedding sin the clause. The more tokens are included in each sentence, the more averaged, ie blurred, the sentence embedding will be. Clusters will be less visible in this blurred landscape, so the fewer tokens the better for finding clusters. Yet, one token per clause is too few, simply the embedding of a single word in isolation from the clause's intent.

The max number of tokens has been hand picked at the 90th percentile (38 tokens).

So, we will also remove the very short clauses, as they may be single words and other unuseful examples. This min_length has been arbitrarily set at 10 tokens. This makes the range, from 10 tokens to 38, fairly narrow, so may assist training as all clauses will be of similar length. Note, there are always at least 2 tokens, start token and finish token, so 10 tokens actually means 8 tokens of text.

In [19]:

```
min_length = 10
max_length = int(np.percentile(token_lengths['train'], 90))

print('Before cropping the clauses...')
print('train data length:', len(sentences_train))
print('testi data length:', len(sentences_testi))
print('valid data length:', len(sentences_valid))

# Get mask for rows where both the cause and effects have a number of tokens between a minimum and the maximum.
# We don't want sentences too Long, neither too short
# Minimum Length has been arbitrarily set to 10 tokens. Max Length is calculated above.

masks={}
for dataset in [sentences_train, sentences_testi, sentences_valid]:
    # create the mask
    masks[dataset.name] = [ min_length <= len(clause) <= max_length for clause in dataset['text_idxs'] ]
    # add the masks to the data and save for future reference
    dataset['mask'] = masks[dataset.name]
    dataset.to_csv(path_or_buf=os.path.join(path, 'sentences_'+dataset.name+'.csv'), index=False, sep='~')
```

Before cropping the clauses...

```
train data length: 434481
testi data length: 24479
valid data length: 25014
```

```
In [20]: # apply those masks to filter out the longer records
# also reset the dataframe indexes, to extract batches we'll benefit from sequential indexes starting at 0
# But we keep them (drop=False) so we can reliably tie the datasets back together.
train = sentences_train[masks['train']].reset_index(drop=False)
testi = sentences_testi[masks['testi']].reset_index(drop=False)
valid = sentences_valid[masks['valid']].reset_index(drop=False)

print('After cropping clauses to min_length=' , min_length, ' tokens, and max_length=' ,max_length,' tokens.')
print('train data length:', len(train))
print('testi data length:', len(testi))
print('valid data length:', len(valid))

After cropping clauses to min_length= 10  tokens, and max_length= 37  tokens.
train data length: 369930
testi data length: 20740
valid data length: 21329
```

Pad Input Tokens

Now we need to apply the padding, so that all token vectors are of length = max_length

The index of the padding token for Albert is 0, we need to pad to max number of indexes for the batch

The final token must always be 102, so the padding 0s must be inserted before that final token. We could have padded whilst using the tokenizer, but we would end up saving huge files full of zeros.

```
In [21]: # We will submit the data to Albert in batches and already padded

def get_batch_for_Albert(tokens_list_batch, max_length=max_length, return_tensors='pt'):
    # batch is a list of lists

    tokens_list_lengths = []

    # apply the padding to max_length
    for i in range(len(tokens_list_batch)):

        tokens_list_length = len(tokens_list_batch[i])

        # get pads to desired max_length
        padding = [0] * (max_length - tokens_list_length)

        # insert pads to list
        tokens_list_batch[i][-1:-1] = padding

        # get list of token lengths,
        # we need this to average the token embeddings into a sentence embedding
        # because we want to exclude the embeddings for 'padding' tokens, these are not useful information.
        # Since final token is always '3', ie the closing token, we can ignore it when averaging tokens into sentences
        tokens_list_lengths = (max_length-1) if tokens_list_lengths == max_length else tokens_list_lengths
        tokens_list_lengths.append(tokens_list_length)

    # create the sentence segment ids, always same length, max_length
    # first in tensorflow format
    if return_tensors == 'tf':

        input_ids      = tf.convert_to_tensor( tokens_list_batch, dtype='int32')
        token_type_ids = tf.convert_to_tensor( np.zeros((input_ids.shape[0], max_length)), dtype='int32' )
        attention_mask = tf.convert_to_tensor( np.where(input_ids > 0, 1, 0),   dtype='int32' )

    # or in pytorch format
    elif return_tensors == 'pt':

        input_ids      = torch.tensor( tokens_list_batch, dtype=torch.int32 )
        token_type_ids = torch.tensor( np.zeros((input_ids.shape[0], max_length)), dtype=torch.int32 )
        attention_mask = torch.tensor( np.where(input_ids > 0, 1, 0),   dtype=torch.int32 )

    else:
        raise Exception('Unknown return_tensors type')

    # get format for Albert
    batch_inputs = { 'input_ids'      : input_ids.to('cuda'),
                    'token_type_ids': token_type_ids.to('cuda'),
                    'attention_mask': attention_mask.to('cuda') }

    return batch_inputs, tokens_list_lengths
```

Let's inspect an example

```
In [22]:
```

14 / 71

Using Albert

Calling `BertModel.from_pretrained()` will fetch the pretrained BERT model from huggingface. When we load the `bert-base-uncased`, we see the definition of the model printed in the logging. The model is a deep neural network with 12 layers. Explaining the layers and their functions is outside the scope of this post, and you can skip over this output for now.

`model.eval()` puts our model in evaluation mode as opposed to training mode. In this case, evaluation mode turns off dropout regularization which is used in training.

```
In [23]: Albert_embedder = AlbertModel.from_pretrained('albert-base-v2')
Albert_embedder.to("cuda")
# Put the model in "evaluation" mode, meaning feed-forward operation (any dropouts will be ignored)
Albert_embedder.eval()
```

```
Out[23]: AlbertModel(  
    (embeddings): AlbertEmbeddings(  
        (word_embeddings): Embedding(30000, 128, padding_idx=0)  
        (position_embeddings): Embedding(512, 128)  
        (token_type_embeddings): Embedding(2, 128)  
        (LayerNorm): LayerNorm((128,), eps=1e-12, elementwise_affine=True)  
        (dropout): Dropout(p=0, inplace=False)  
    )  
    (encoder): AlbertTransformer(  
        (embedding_hidden_mapping_in): Linear(in_features=128, out_features=768, bias=True)  
        (albert_layer_groups): ModuleList(  
            (0): AlbertLayerGroup(  
                (albert_layers): ModuleList(  
                    (0): AlbertLayer(  
                        (full_layer_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)  
                        (attention): AlbertAttention(  
                            (query): Linear(in_features=768, out_features=768, bias=True)  
                            (key): Linear(in_features=768, out_features=768, bias=True)  
                            (value): Linear(in_features=768, out_features=768, bias=True)  
                            (attention_dropout): Dropout(p=0, inplace=False)  
                            (output_dropout): Dropout(p=0, inplace=False)  
                            (dense): Linear(in_features=768, out_features=768, bias=True)  
                            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)  
                        )  
                        (ffn): Linear(in_features=768, out_features=3072, bias=True)  
                        (ffn_output): Linear(in_features=3072, out_features=768, bias=True)  
                        (dropout): Dropout(p=0, inplace=False)  
                    )  
                )  
            )  
        )  
    )  
    (pooler): Linear(in_features=768, out_features=768, bias=True)  
    (pooler_activation): Tanh()  
)
```

Next, let's use Albert on our example text, and fetch the hidden states of the network!

Side note: `torch.no_grad` tells PyTorch not to construct the compute graph during this forward pass (since we won't be running backprop here)—this just reduces memory consumption and speeds things up a little.

```
In [24]: # Run the text through BERT, and collect all of the hidden states produced
# from all 12 layers.
with torch.no_grad():

    outputs = Albert_embedder(**test_batch_inputs)

    # Evaluating the model will return a different number of objects based on
    # how it's configured in the `from_pretrained` call earlier. In this case,
    # because we set `output_hidden_states = True`, the third item will be the
    # hidden states from all layers. See the documentation for more details:
    # https://huggingface.co/transformers/model_doc/bert.html#bertmodel
    last_hidden_states = outputs.last_hidden_state

    # Let's view those hidden states
    print("Hidden state shape:")
    print("[batch_size, tokens, dims_per_token]")
    print(last_hidden_states.shape)
    print()
    print("Hidden states:")
    print(last_hidden_states)
```

```
Hidden state shape:
[batch_size, tokens, dims_per_token]
torch.Size([2, 37, 768])

Hidden states:
tensor([[[ 0.6911,  0.8582,  1.8561, ...,  0.0467,  0.2222,  0.2539],
       [-0.1972, -0.3015,  0.5777, ...,  0.0838,  0.7703, -2.2283],
       [-0.3768, -0.0954,  0.7758, ..., -0.4187, -0.3720, -0.0363],
       ...,
       [ 0.0545,  0.1338, -0.0559, ..., -0.0717,  0.1268,  0.1909],
       [ 0.0545,  0.1338, -0.0559, ..., -0.0717,  0.1268,  0.1909],
       [ 0.0546,  0.1336, -0.0561, ..., -0.0716,  0.1268,  0.1908]],

      [[-0.6589,  0.8848,  1.7386, ..., -1.4852,  1.2754, -0.3603],
       [-0.0824,  0.1646, -0.0420, ...,  1.0562,  0.3428, -0.1525],
       [-0.3175, -0.4258,  0.1213, ...,  0.1023, -0.1616, -0.8059],
       ...,
       [ 0.0914,  0.1099, -0.0822, ..., -0.1133,  0.1550,  0.1829],
       [ 0.0914,  0.1099, -0.0822, ..., -0.1133,  0.1550,  0.1829],
       [ 0.0915,  0.1098, -0.0824, ..., -0.1132,  0.1549,  0.1828]]],  
device='cuda:0')
```

Creating word and sentence vectors from hidden states

In previous work we have extracted the final four layers of the BERT model, each of which has an activation for each token. These were then summed to get an embedding vector representing each token.

On this occasion we need an embedding for the entire sentence. The recommended way forward is to extract the activations for the final hidden layer of the Albert model. This yields a vector for each token. Those are then averaged into one embedding vector for the entire sentence. We exclude the embeddings of padding tokens from the averaging.

Alternative approaches may also be valid.

A Single Function to Get Albert Embeddings

Let's put these functions together for getting a sentence embedding and an embedding for just the searchword.

- grab a batch of 32 records
- get embeddings from Albert
 - for the sentence, average the tokens, excluding padding, save the embedding for the sentence (768 dims)
 - for the searchword, find the relevant token and save its embedding separately (also 768 dims)

```
In [25]: # function of functions

def get_embeddings(batch, batch_args, Albert_embedder):

    # get tokens (padded to max_length) and segment ids for the batch
    batch_inputs, tokens_list_lengths = get_batch_for_Albert(batch)

    # submit those tokens and segment ids to Albert
    # get the final layer, which is the most 'contextualised', as we'd expect and seek.
    with torch.no_grad():
        outputs = Albert_embedder(**batch_inputs)
        hidden_states = outputs.last_hidden_state.cpu()

    # torch tensor dims are now [batch_size, tokens, features]
    # convert to numpy
    embeddings_np = hidden_states.numpy()

## SEARCHWORD EMBEDDINGS
```

```

# the searchword's location in each batch is a list, presented as the batch_args
# change batch_args to numpy
batch_args = np.array(batch_args)
# ensure this was submitted as correct shape
assert len(batch_args.shape) == 1, 'Too many dims for batch_args. Should be submitted as 1D list'
# ensure batch_args has as many dims as embeddings (ie 3)
batch_args = batch_args[:, np.newaxis, np.newaxis]
# select a token's embeddings from each sentence's embedding, ie, select on the 'token' dimension, axis = 1
# return shape [batch_size, 1, features]
embeddings_searchword = np.take_along_axis(embeddings_np, batch_args, axis=1)
# squeeze singleton dimension
embeddings_searchword = np.squeeze(embeddings_searchword)

## SENTENCE EMBEDDINGS
# for each row (ie sentence) average the values across the non padding tokens
the_range      = range(len(tokens_list_lengths))
embeddings_ragged = [ np.array(embeddings_np[row, 1:length, :]) 
                      for row, length in zip(the_range, tokens_list_lengths) ]

# average over all tokens to get an embedding for the entire sentence
embeddings_sentence = np.array([ np.mean(embeddings_ragged[row], axis=0) for row in the_range] )

# return the embeddings, shape is [batch_size, features]
return embeddings_sentence, embeddings_searchword

```

Let's run this on a sample of 1000 embeddings, how long does it take?

This is pertinent because we have approx 400k sentences to analyze

In [26]:

```

sample_testi      = testi[['text_idxs', 'searchword_tkn_idx']].sample(n=1000)
batch_testi       = sample_testi['text_idxs'].tolist()
batch_args_testi = sample_testi['searchword_tkn_idx'].tolist()

# we only want the first item from each searchword list
batch_args_testi = [tkn_list[0] for tkm_list in batch_args_testi]

# time the embedding task
start_time = time.time()

# get the embeddings by submitting the tokens to the BERT model, returns np.array
batch_embedding_sent, batch_embedding_sw = get_embeddings(batch           = batch_testi,
                                                       batch_args      = batch_args_testi,
                                                       Albert_embedder= Albert_embedder)

# get time
print('Time to extract 1000 embeddings= %s seconds' % (time.time() - start_time))
print('\n')
# print results of example to screen
print('Shape of batch of sentence embeddings [batch,features]: ', batch_embedding_sent.shape)
print('Shape of batch of searchword embeddings [batch,features]: ', batch_embedding_sw.shape)
print('\n')
# print example embedding
print('Example, first 10 values of a sentence embedding')
print(batch_embedding_sent[0,0:10])
print('\n')
print('Example, first ten values of a searchword embedding')
print(batch_embedding_sw[0,0:10])

```

Time to extract 1000 embeddings= 1.7652759552001953 seconds

Shape of batch of sentence embeddings [batch,features]: (1000, 768)
Shape of batch of searchword embeddings [batch,features]: (1000, 768)

Example, first 10 values of a sentence embedding
[0.17961866 -0.5665813 1.2265397 -0.15601915 0.63158953 0.49013734
 0.32842267 0.02006856 0.4823007 0.72329676]

Example, first ten values of a searchword embedding
[-0.13280717 -1.2974725 1.2157118 -1.6207521 0.16532004 1.0643159
 1.2250993 -1.8797677 0.39308518 -0.1445304]

In [113...]

```

# 2 seconds for 1000 embeddings.
# 350 * 2s = 700s, approx 12mins to get all the sentence embeddings for the training set alone.

from sklearn.preprocessing import StandardScaler

batch_size    = 1000

for dataset_pair in [[train, 'train'], [testi, 'testi'], [valid, 'valid']] :
    start_time = time.time()
    dataset, dataset_name = dataset_pair

```

```

# trim dataset to required columns only
dataset      = dataset[['text_idxs', 'searchword_tkn_idx']]
batch_num    = math.ceil(len(dataset)/batch_size)
batch_start  = 0

if batch_size > len(dataset):
    raise Exception('batch size larger than data')

for i in tq.tqdm(range(batch_num)):

    # get batch start/end, capped at file length.
    batch_end = min(batch_start + batch_size, len(dataset))

    # get batch_tokens as series from pandas
    batch      = dataset.iloc[batch_start:batch_end].copy()
    batch_tokens = batch['text_idxs'].tolist()
    batch_args  = batch['searchword_tkn_idx'].tolist()
    batch_args  = [tkn_list[0] for tkn_list in batch_args]

    # get the embeddings by submitting the tokens to the BERT model, returns np.arrays
    embeddings_sentence_bt, embeddings_serchwrd_bt = get_embeddings(batch           = batch_tokens,
                                                                    batch_args     = batch_args,
                                                                    Albert_embedder = Albert_embedder)

    # stack the embeddings
    if i == 0:
        embeddings_sentence = embeddings_sentence_bt
        embeddings_serchwrd = embeddings_serchwrd_bt
    else:
        # stack the data
        embeddings_sentence = np.vstack((embeddings_sentence, embeddings_sentence_bt))
        embeddings_serchwrd = np.vstack((embeddings_serchwrd, embeddings_serchwrd_bt))

    # increment batch
    batch_start = batch_end

# end for loop

# Scale each column of data
# Note this scales the data for the PCA and autoencoder work, so each dim is comparable in size to any other
# It will not make data more 'gaussian', that task follows in later code chunks
scaler_sentence = StandardScaler().fit(embeddings_sentence)
scaler_serchwrd = StandardScaler().fit(embeddings_serchwrd)

embeddings_sentence = scaler_sentence.transform(embeddings_sentence)
embeddings_serchwrd = scaler_serchwrd.transform(embeddings_serchwrd)

# save the embeddings as numpy
np.save(file=os.path.join(path, 'embeddings_sentence_'+dataset_name+'.npy'), arr=embeddings_sentence)
np.save(file=os.path.join(path, 'embeddings_serchwrd_'+dataset_name+'.npy'), arr=embeddings_serchwrd)

# it would be convenient if those embeddings were part of the pandas file with text and tokens
# but that results in huge files which are causing my python kernel to crash!

# how long did that take?
print('Time to extract and save', dataset_name, 'embeddings = ', str((time.time() - start_time)/3600.00), 'hrs')

```

Time to extract and save train embeddings = 0.18812816361586251 hrs

Time to extract and save testi embeddings = 0.00955253091123369 hrs

Time to extract and save valid embeddings = 0.009841903249422709 hrs

In [114...]

```
train_embed = np.load(os.path.join(path, 'embeddings_sentence_train.npy'))
train_embed.shape
```

Out[114...]

Sentence Embeddings with Sbert

The method used above to extract sentence embeddings is an extraction of embeddings from first principles, it is a lot of work and it is not optimised. Its advantage is that the transformer can be transfer trained to any dataset and the embeddings are then specific to a given dataset. A useful technique to know.

Optimised and high quality sentence embeddings are possible using 'sbert', which is much simpler to use. Although, we cannot get embeddings for contextualised words from it, ie the contextualised searchword. We would have to settle for one embedding for each searchword, the same in all contexts.

Sbert follows the paper from TU Darmstadt at [<https://arxiv.org/pdf/1908.10084.pdf>]. The package is available from [<https://www.sbert.net/>]

The below code is provided as reference, to show how simple it is to use Sbert, as compared with acquiring our own embeddings. Unfortunately, the work by Zhu & deMelo 2020 found Sbert embeddings to be particularly poor for finding semantic analogies. Therefore they are not used in

this work.

In [116...]

```
from sentence_transformers import SentenceTransformer
model_sbert = SentenceTransformer('paraphrase-mpnet-base-v2', device='cuda')
# get sentence embeddings for all datasets, no need to batch.
# note, resulting embeddings have 768 features, same as above model
for dataset_pair in [[train, 'train'], [testi, 'testi'], [valid, 'valid']] :
    dataset, dataset_name = dataset_pair
    sentences_sbert = dataset['text'].tolist()
    embeddings_sbert = model_sbert.encode(sentences_sbert)
    np.save(file=os.path.join(path, 'embeddings_sbert_'+dataset_name+'.npy'), arr=embeddings_sbert)
```

Implementing Approach 1

Each clause or sentence is now represented by a vector with 768 values, aka an embedding. That's very high dimensionality data, we need to reduce that dimensionality for practical processing purposes and to aid clustering. 768 dims would certainly suffer from the 'curse of high dimensionality', whereby each additional dimension increases the potential distance between data points. For example, points in 1D space which are then projected into a 3D space will find they have enormous opportunity to be spread out in the additional two dimensions. Simple euclidean distance is not an appropriate tool for analysing space with many more than 3 dimensions. So tools like K-means are not entirely appropriate.

First we need to reduce those 768 dimensions to something manageable for clustering. PCA (principal component analysis) is the most common tool for this task. However, autoencoders are better at compressing non-linear data than PCA. Activations from Albert are likely to be non-linear. So we will use an autoencoder and compare its data compression performance with PCA by comparing the reconstruction loss of the two methods. It is expected that we can achieve better than 10:1 compression ratio, so compress the 768 dimensions into approx 60 dimensions.

The autoencoder will need to be optimised, which means a hyper-parameter search, a task best conducted by a cluster of machines working in parallel. AzureML will be used for this.

Even 60 dims may remain too many dimensions for clustering a third of a million points in a reasonable amount of time on a desktop machine. We can further prepare the data for clustering by using UMAP. This is a tool like t-SNE which attempts to reduce dimensions whilst maintaining distance between points. Autoencoders and PCA aim to retain a large part of the information in the data. Whereas UMAP only aims to retain the relative distance between points, which is sufficient for clustering. In this way, we will further reduce the dimensions by half.

Finally, we can turn to clustering algorithms. Even with approx 30 dims, our data remains high dimensional data. The appropriate tool for identifying and quantifying the clusters will likely be HDBSCAN. There is a good explanation of high dimensionality clustering algorithms at:

- <https://towardsdatascience.com/how-to-cluster-in-high-dimensions-4ef693bacc6>

Finally, we will want to visualise our clusters. This will mean projecting the high dimensionality data down to just two dimensions. PCA will not cope as there will be too many dimensions and too little significance represented by the top 2 dims. This is another task for UMAP.

In [117...]

```
# confirm we can load those embedding files
def get_pandasfromcsv(filename, name):
    data = pd.read_csv(os.path.join(path,filename), sep='~')
    data['text_idxs'] = [literal_eval(text_idxs) for text_idxs in data['text_idxs']]
    data.name = name
    return data

embeddings_train = np.load(os.path.join(path, 'embeddings_sentence_train.npy'))
embeddings_testi = np.load(os.path.join(path, 'embeddings_sentence_testi.npy'))
embeddings_valid = np.load(os.path.join(path, 'embeddings_sentence_valid.npy'))
```

What is the Distribution of the Embedding Data?

We are proposing to pass the standardised embeddings into a autoencoder. This is a deep learning network and will work best if the data is approximately normally distributed.

So, first let's randomly select a dimension of the embeddings and view a QQ plot. Then we will test all 768 dimensions to see if they are all approximately normal. We'll use the Shapiro-Wilk test, which gives a probability of 'normalness', higher is better.

In [118...]

```
# The above chart looks good, a randomly selected dimension is indeed normally distributed.
# What about all the other dims?

# seed the random number generator
np.random.seed(20210406)

shapiro_results = []
for dim in range(768):
```

```

# normality test
stat, p = ss.shapiro(embeddings_valid[:,dim])

# interpret
alpha = 0.05
if p > alpha:
    # Sample Looks Gaussian (fail to reject H0)
    interpretation = 'Gaussian'
else:
    # Sample does not Look Gaussian (reject H0)
    interpretation = 'Not Gaussian'

result = {'dim':dim, 'interpretation':interpretation, 'p_value':p}

shapiro_results.append(result)

# convert to pandas for convenience
shapiro_results = pd.DataFrame(shapiro_results)

# display top 10 least 'normal' dims
shapiro_results.sort_values(by='p_value', ascending=True).head(10)

```

p-value may not be accurate for N > 5000.

Out[118...]

	dim	interpretation	p_value
318	318	Not Gaussian	0.0
248	248	Not Gaussian	0.0
668	668	Not Gaussian	0.0
651	651	Not Gaussian	0.0
649	649	Not Gaussian	0.0
101	101	Not Gaussian	0.0
684	684	Not Gaussian	0.0
96	96	Not Gaussian	0.0
122	122	Not Gaussian	0.0
564	564	Not Gaussian	0.0

In [121...]

```

# how many dims are gaussian and how many are not?

print('There are this many Non Gaussian dims :',
      shapiro_results[ shapiro_results['interpretation']=='Not Gaussian' ]['interpretation'].count()
      )

print('There are this many Gaussian dims :',
      shapiro_results[ shapiro_results['interpretation']=='Gaussian' ]['interpretation'].count()
      )

```

There are this many Non Gaussian dims : 741
 There are this many Gaussian dims : 27

In [122...]

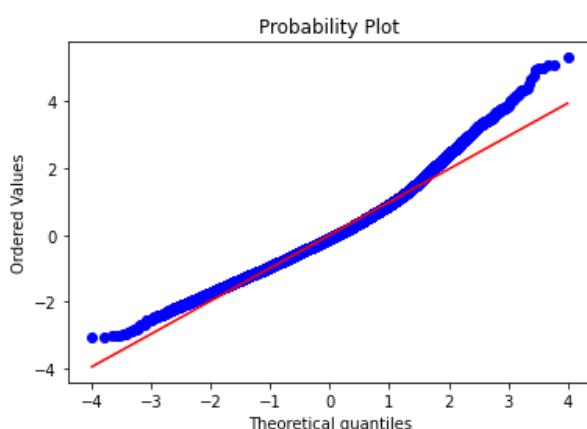
```

# Let's do the QQ chart on that Least gaussian of dimensions
# Looking for a straight line if the data is perfectly normal

least_gaussian_dim = shapiro_results.sort_values(by='p_value', ascending=True).iloc[0,0]

ss.probplot(embeddings_valid[:,least_gaussian_dim], dist="norm", plot=pylab)
pylab.show()

```



In [123...]

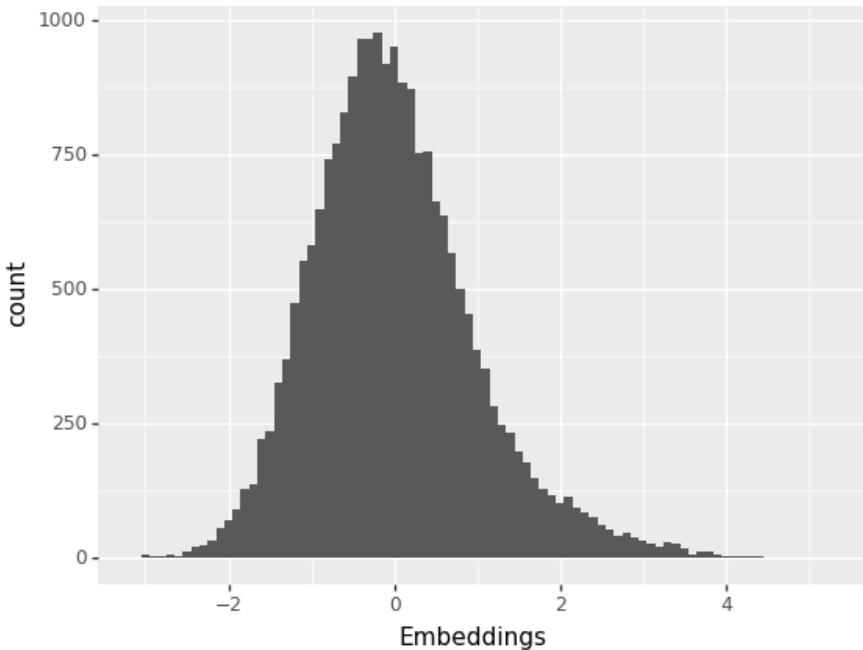
```
# Let's take the Least 'gaussian' and view the distribution ourselves
```

```

hist_data = pd.DataFrame({'Embeddings':embeddings_valid[:,least_gaussian_dim]})

(
    p9.ggplot(hist_data, p9.aes(x='Embeddings'))
    + p9.geom_histogram(binwidth=0.1)
)

```



```
Out[123... <ggplot: (8791900207017)>
```

Yeo Johnson

The above histogram shows a skew is clearly visible, the left hand tail is larger than the right. But, this is the worst example and much less 'normal' data is used in many systems.

Nevertheless, can we improve it?

When we have data which is normal-like, but needs coaxing into normalcy, then the Yeo Johnson transform may help. Box Cox is also an option, but only where data is entirely +ve, embeddings are not.

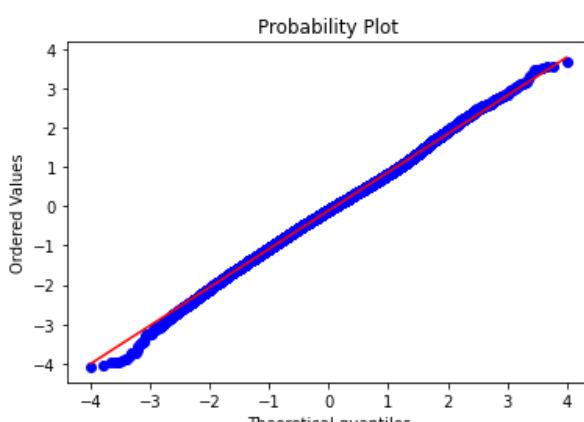
Can Yeo Jonhson help us with the 'least gaussian' of the dimensions? If so, then we can apply it to all dims.

```

In [124... # transform the least gaussian dimension using yeo johnson
embeddings_valid_least_gaussian_yj, lmbda = ss.yeojohnson(embeddings_valid[:,least_gaussian_dim])

# plot QQ for the transformed data
ss.probplot(embeddings_valid_least_gaussian_yj, dist="norm", plot=pylab)
pylab.show()

```

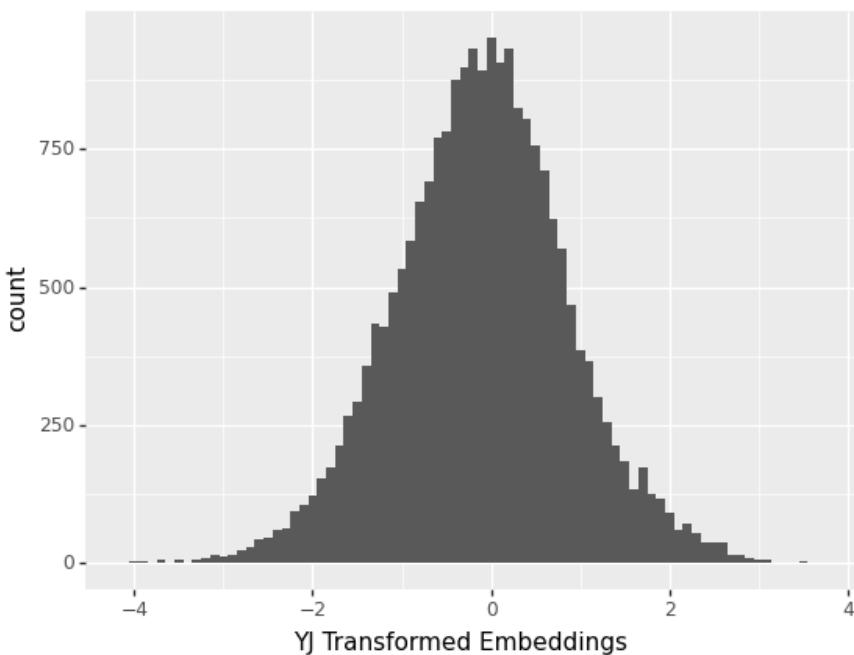


```

In [125... hist_data = pd.DataFrame({'YJ Transformed Embeddings':embeddings_valid_least_gaussian_yj})

(
    p9.ggplot(hist_data, p9.aes(x='YJ Transformed Embeddings'))
    + p9.geom_histogram(binwidth=0.1)
)

```



Out[125... <ggplot: (8791900206987)>

In [126...]

```
# another measure is to repeat the Shapiro-Wilks test, before and after Yeo Johnson...
print('Before Yeo Johnson, the least gaussian dimension had this Shapiro score...')
print(ss.shapiro(embeddings_valid[:,least_gaussian_dim]))
print()
print('After Yeo Johnson, the least gaussian dimension has this Shapiro score...')
print(ss.shapiro(embeddings_valid_least_gaussian_yj))
```

Before Yeo Johnson, the least gaussian dimension had this Shapiro score...
ShapiroResult(statistic=0.9755314588546753, pvalue=0.0)

After Yeo Johnson, the least gaussian dimension has this Shapiro score...
ShapiroResult(statistic=0.9985800385475159, pvalue=7.426849391554968e-13)
p-value may not be accurate for N > 5000.

Yeo Johnson makes Small Improvement

729 of the 768 dims in the validation data appear to be non gaussian. There is an argument for applying Yeo Johnson to all of them. The QQ plot and histogram show the YJ transform to be successful, at least to the human eye. Whereas, the Shapiro score improves only slightly.

Since the shapiro score improves only slightly, we could probably proceed without applying the YJ transform. However, every advantage will be helpful when we later try to compress such high dimensional data into as few dimensions as possible, not least when using PCA. Therefore, we will apply YJ to all dims.

In [127...]

```
# create function to transform all dims in a dataset
def apply_YJ(embeddings):

    embeddings_yj_list = []

    for dim in tq.tqdm(range(embeddings.shape[1])):
        embeddings_yj, lmbda = ss.yeojohnson(embeddings[:,dim])
        embeddings_yj_list.append(embeddings_yj.tolist())

    embeddings_yj = np.array(embeddings_yj_list).T

    return embeddings_yj

# apply the function to all datasets
embed_pairs = [(embeddings_train, 'embeddings_sentence_train_yj.npy'),
                (embeddings_testi, 'embeddings_sentence_testi_yj.npy'),
                (embeddings_valid, 'embeddings_sentence_valid_yj.npy')]

# save the results to new files
for embeds, filename in embed_pairs:
    embeddings_yj = apply_YJ(embeds)
    np.save(file=os.path.join(path, filename), arr=embeddings_yj)
```

In [2]:

```
# confirm we can load those transformed embedding files, 8.3GB for training alone
```

```
# We will Load them as the main embedding files, for use throughout the clustering work

embeddings_train_yj = np.load(os.path.join(path, 'embeddings_sentence_train_yj.npy'))
embeddings_testi_yj = np.load(os.path.join(path, 'embeddings_sentence_testi_yj.npy'))
embeddings_valid_yj = np.load(os.path.join(path, 'embeddings_sentence_valid_yj.npy'))

# we still have use for the raw, unprocessed, embeddings

embeddings_train = np.load(os.path.join(path, 'embeddings_sentence_train.npy'))
embeddings_testi = np.load(os.path.join(path, 'embeddings_sentence_testi.npy'))
embeddings_valid = np.load(os.path.join(path, 'embeddings_sentence_valid.npy'))
```

PCA

Investigate the performance of PCA in reducing the number of dimensions.

Will later compare with autoencoder performance, using reconstruction error as the metric

```
In [130...]
# Investigate performance of PCA, to reduce the huge number of dimensions (768) down to something more manageable for clustering

# fit to training set (may be different for test and validation sets)
# Note, embeddings have already been standardised, which would otherwise be a necessary step before PCA
pca_yj = PCA()
pca_yj.fit(embeddings_train_yj)

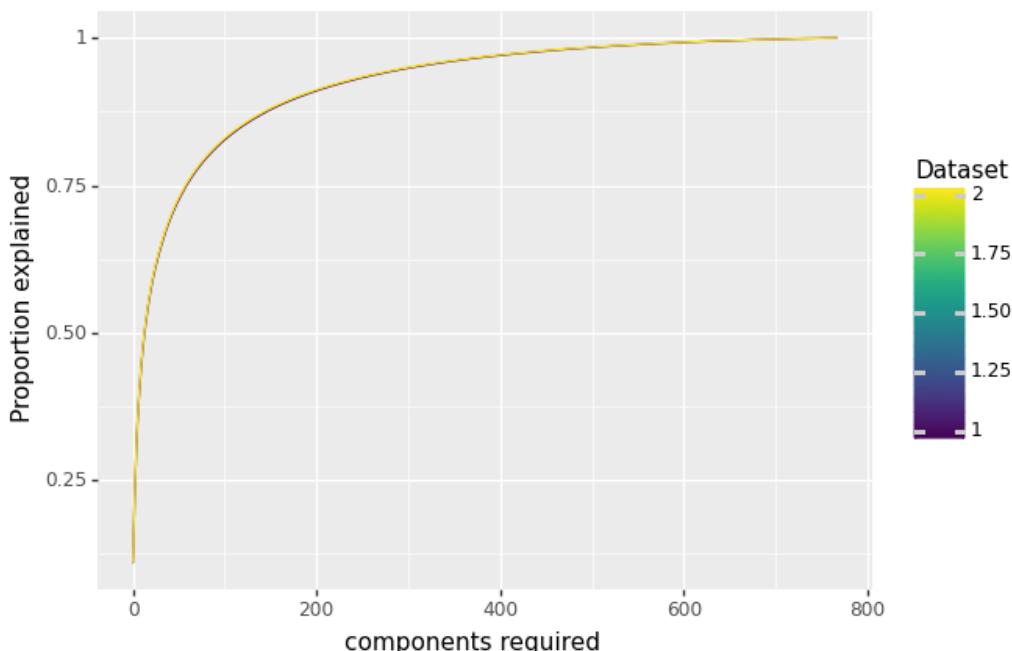
# repeat for unprocessed embeddings, so we can compare PCA's performance on the two datasets
pca_un = PCA()
pca_un.fit(embeddings_train)

# get proportion of variance explained vs dimensions retained for yeo johnson data. Then flag them as dataset
pca_cumsum_yj = np.cumsum(pca_yj.explained_variance_ratio_[:,np.newaxis])
pca_cumsum_yj = np.hstack( (np.zeros((pca_cumsum_yj.shape[0],1)), # dataset
                           np.arange(pca_cumsum_yj.shape[0])[:,np.newaxis], # number of dimensions
                           pca_cumsum_yj ) ) # proportion of variance explained

# repeat on unprocessed figures but for unprocessed data. Flag them as dataset 1
pca_cumsum_un = np.cumsum(pca_un.explained_variance_ratio_[:, np.newaxis])
pca_cumsum_un = np.hstack( (np.ones((pca_cumsum_un.shape[0],1)), # dataset
                            np.arange(pca_cumsum_un.shape[0])[:, np.newaxis], # number of dimensions
                            pca_cumsum_un ) ) # proportion of variance explained

# bring them together
pca_cumsum = np.vstack((pca_cumsum_yj, pca_cumsum_un))
pca_cumsum = pd.DataFrame({'Dataset':pca_cumsum[:,0]+1,
                           'Dimensions': pca_cumsum[:,1],
                           'VarianceExplained':pca_cumsum[:,2]})
```

```
In [131...]
# plot the PCA Variance Explained vs Dimensions for both datasets, yeo johnson and unprocessed
(
    p9.ggplot(pca_cumsum)
    + p9.aes(x='Dimensions', y='VarianceExplained', color='Dataset', group='Dataset')
    + p9.geom_line()
    + p9.labs(x='components required', y='Proportion explained')
)
```



Out[131... <ggplot: (8792560199135)>

The two lines overlap. Yeo Johnson transform has made no appreciable difference to performance of PCA.

So it will NOT be applied, it is an unnecessary additional step. We will proceed with the unprocessed embeddings

```
In [132... # The 'knee' of this chart is at 0.8, good ol 80/20 rule
# so to separate the signal from the noise we can select the components which give us 80% of the variance
pca_80 = PCA(0.8)
pca_80.fit(embeddings_train)

embeddings_train_80 = pca_80.transform(embeddings_train)
print("Shape of embeddings : ", embeddings_train_80.shape)
print("Hence, dimensions required : ", embeddings_train_80.shape[1])
```

Shape of embeddings : (369930, 82)
Hence, dimensions required : 82

Autoencoder

As discussed, autoencoders have a reputation for being better than PCA at compressing non-linear relationships within data. However, they are more 'opaque' than PCA. We cannot ask what proportion of variance they retain. We can only ask about the reconstruction errors, although that is a metric we can also use with PCA.

Autoencoders come in many flavours, we'd like a simple model where the latent_space is sufficiently structured that clusters plotted within that space are meaningful. A variational autoencoder achieves this, this is typically developed for 'generative' modelling, but we want to use it for its structured latent space. A plain autoencoder may achieve the same results, but the VAE at least imposes the principle of a structured (meaningful) latent space.

There are a number of hyperparameters applicable to an autoencoder, amongst them are:

- Number of layers
- Number of nodes in each layer
- Latent space size (ie final number of dimensions)
- Batch size

When training the model it will be useful to explore these hyperparameters and select the best combination. This is where AzureML's cluster compute service is useful, in training models in parallel thus saving time when optimising hyperparameters.

Use Azure ML

Hyperparameter searches require training the model many times, which is very time consuming. It is effective to purchase compute power and run the experiments in parallel.

Notes on AzureML environments

Microsoft AzureML allows us to write notebooks like this one, but execute selected code, or entire notebooks, on cloud compute. Microsoft offer individual servers for this task, or clusters whereby a number of experiments can be run in parallel.

Microsoft also offer 'AutoML', which will take the data presented and automatically apply a number of common models on that data in order to discover which is most appropriate on whatever metric you choose. It also automatically preprocesses the data to achieve this, so is a real time saver.

AutoML typically applies statistical learning models, such as LightGBM, to Classification. AutoML can also be used for Regression or Time Series forecasting. However, there is not yet an AutoML tool for clustering or dimensionality reduction, which are the objective of this notebook. So we will be using AzureML for its compute power and giving it a python model to execute using various different hyperparams.

The following code is adapted from examples at:

<https://github.com/Azure/MachineLearningNotebooks/blob/master/how-to-use-azureml/ml-frameworks/keras/train-hyperparameter-tune-deploy-with-keras/train-hyperparameter-tune-deploy-with-keras.ipynb>

and

<https://docs.microsoft.com/en-us/azure/machine-learning/how-to-train-tensorflow>

Cheatsheets can be found at: <https://azure.github.io/azureml-examples/docs/cheatsheet/data/>

AzureML Workspaces

A workspace is simply a collection of the resources required for an experiment; datastore, dataset, compute, models etc.

AzureML Datastores

The data must be uploaded to the cloud, where it is typically stored in an AzureBlob, although cloud SQL and NoSQL databases are also possible. That blob or database is called a 'datastore'. Its use is chargeable.

```
In [3]: # get the default workspace for our AzureML subscription

## Create Workspace or Load default from config file
print('Ready to use Azure ML {} {}'.format(azureml.core.VERSION))

# We can Load the default workspace with the below code, which reads a Local file with the subscription details
# NOTE: we will later delete the entire workspace in order to prevent Azure storage charges
# Therefore, the workspace file has been amended with a specific workspace name, not used on any other projects

try:
    ws = Workspace.get( name          = 'Autoenc_ws',
                        subscription_id = 'GUID',
                        resource_group  = 'AzureML_PAYG')

except:
    print('Workspace does not yet exist, will be created')

    ws = Workspace.create( name          = 'Autoenc_ws',
                           subscription_id = 'GUID',
                           resource_group  = 'AzureML_PAYG',
                           create_resource_group= False,
                           location        = 'uksouth')

# Normally we wouldn't create a whole new workspace,
# but it does make deletion simple, thus preventing unexpected Azure charges
# Instead, we would select the default workspace using a config file stored locally
# Below is the code for doing this, where the config file refers to the above workspace
# ws = Workspace.from_config()

# The config file is in json file and is very simple:
#{ "subscription_id": "Your subscription GUID goes here",
#  "resource_group": "AzureML_PAYG",
#  "workspace_name": "Your unique workspace name" }

# Let's confirm the workspace details
print('Workspace name : ' + ws.name,
      'Azure region : ' + ws.location,
      'Resource group : ' + ws.resource_group, sep='\n'
      '# Subscription id : ' + ws.subscription_id
      )

# set the datastore to which we upload data, we'll use the default for the workspace
datastore = ws.get_default_datastore()
print('Default datastore: ' + datastore.name)
```

```
Ready to use Azure ML 1.26.0
Workspace name : Autoenc_ws
Azure region : uksouth
Resource group : AzureML_PAYG
Default datastore: workspaceblobstore
```

```
In [134... ##### ENSURE THE EMBEDDINGS ARE IN A FOLDER NAMED 'AZUREML'
## This has not been coded, but just copy relevant files to such a folder, then run this code.

# upload data to the default AzureML datastore
# Does not upload single file, must be entire folder
datastore.upload(src_dir  = os.path.join(path, 'AzureML'),
                 target_path = 'embedding_wants',
                 overwrite   = True)
```

```
Uploading an estimated of 4 files
Uploading /home/oliver/Documents/Deep/Data/AzureML/enc_umap/embeddings_train_un_enc_umap_30.npy
Uploaded /home/oliver/Documents/Deep/Data/AzureML/enc_umap/embeddings_train_un_enc_umap_30.npy, 1 files out of an estimated total of 4
Uploading /home/oliver/Documents/Deep/Data/AzureML/embeddings_sentence_valid.npy
Uploaded /home/oliver/Documents/Deep/Data/AzureML/embeddings_sentence_valid.npy, 2 files out of an estimated total of 4
Uploading /home/oliver/Documents/Deep/Data/AzureML/embeddings_sentence_testi.npy
Uploaded /home/oliver/Documents/Deep/Data/AzureML/embeddings_sentence_testi.npy, 3 files out of an estimated total of 4
Uploading /home/oliver/Documents/Deep/Data/AzureML/embeddings_sentence_train.npy
Uploaded /home/oliver/Documents/Deep/Data/AzureML/embeddings_sentence_train.npy, 4 files out of an estimated total of 4
Uploaded 4 files
```

```
Out[134... $AZUREML_DATAREFERENCE_bd86514fb35b40d4b1c50f7dfa43523a
```

AzureML Datasets

The files in the datastore can be used to form a 'datasets'. A 'dataset' is a reference to data - either in a datastore or behind a public URL.

Datasets must be 'registered' with the workspace to enable their use in experiments in the workspace. This also gives us an opportunity to assign them tags, for example a tag to set the version of the data.

Datasets can be 'tabular' or 'file'. Typically, tabular data is in CSV format, where it can easily be inspected online and used for AutoML. Whereas, file data might be images for image recognition training.

However, CSV format does not suit all tabular data. The numpy file of embeddings, as used in this analysis, is 3.1GB. When saved into CSV format using np.savetxt() it expands to 9GB. That takes much longer to upload to the cloud. Furthermore, having been uploaded it must be converted, by the notebook, back to numpy format, which takes substantial time.

So, our embedding data is tabular, but we will define it as a file. This doesn't matter since we do not intend to use autoML.

```
In [4]: # create a FileDataset around the directory of the datastore,
# use wildcard to select all (train, test, valid) numpy files into the new dataset
embeddings_dataset = Dataset.File.from_files(path=(datastore, 'embedding_wants'))
```



```
In [5]: # register the data
dataset_registered = False

try:
    # can we get the registered dataset by its name?
    temp = Dataset.get_by_name(workspace = ws, name = 'embedding-wants-dataset')
    dataset_registered = True

except:
    print("The dataset 'embedding-wants-dataset' is not registered in workspace yet.")

if not dataset_registered:
    # Register the dataset, with a name
    # Use the register() method to register datasets to your workspace so they can be shared with others,
    # reused across various experiments, and referred to by name in your training script.
    embeddings_dataset = embeddings_dataset.register(workspace
                                                    = ws,
                                                    name
                                                    = 'embedding-wants-dataset',
                                                    description
                                                    = 'training, valid and test dataset',
                                                    tags
                                                    = {'format':'npy'},
                                                    create_new_version=True)

# confirm the datasets we have now registered

print("Datasets Versions:")
for dataset_name in list(ws.datasets.keys()):
    dataset = Dataset.get_by_name(ws, dataset_name)
    print("\t", dataset.name, 'version', dataset.version)
```



```
Datasets Versions:
embedding-wants-enc-umap version 1
embedding-wants-dataset version 1
```

AzureML Compute

The computing power to execute the experiment is defined either as an 'instance' (ie a single virtual server) or as a 'cluster' (a group of virtual servers for executing experiments in parallel)

The compute must be defined prior to executing the experiment, you pay per hour of usage. GPU based systems for deep learning are substantially more expensive, around £1.50/hr per instance.

We will setup a cluster to execute the hyperparam search. This has been done manually in the ml.azure.com, the online studio. A cluster machine with the following equipment has been specified:

- STANDARD_NC6 (6 Cores, 56 GB RAM, 380 GB Disk)
- nvidia Tesla K80 GPU

```
In [6]: # choose a name for your cluster
cluster_name = 'Hyperdrive-01'

# If compute already exists, use it, else create
try:
    training_cluster = ComputeTarget(workspace=ws, name=cluster_name)
    print('Found existing compute target')

except ComputeTargetException:
    print('Creating a new compute target...')
    compute_config = AmlCompute.provisioning_configuration(vm_size = 'STANDARD_NC6',
                                                          max_nodes= 4)
    # create the cluster
    training_cluster = ComputeTarget.create(ws, cluster_name, compute_config)

# can poll for a minimum number of nodes and for a specific timeout.
```

```
# if no min node count is provided it uses the scale settings for the cluster
training_cluster.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)
```

```
# use get_status() to get a detailed status for the current cluster.
print(training_cluster.get_status().serialize())
```

```
Found existing compute target
```

```
Succeeded
```

```
AmlCompute wait for completion finished
```

```
Minimum number of nodes requested have been provisioned
```

```
{'currentNodeCount': 0, 'targetNodeCount': 0, 'nodeStateCounts': {'preparingNodeCount': 0, 'runningNodeCount': 0, 'idleNodeCount': 0, 'unusableNodeCount': 0, 'leavingNodeCount': 0, 'preemptedNodeCount': 0}, 'allocationState': 'Steady', 'allocationStateTransitionTime': '2021-07-06T20:03:27.266000+00:00', 'errors': None, 'creationTime': '2021-04-23T11:27:41.257160+00:00', 'modifiedTime': '2021-04-23T11:27:56.864984+00:00', 'provisioningState': 'Succeeded', 'provisioningStateTransitionTime': None, 'scaleSettings': {'minNodeCount': 0, 'maxNodeCount': 4, 'nodeIdleTimeBeforeScaleDown': 'PT120S'}, 'vmPriority': 'Dedicated', 'vmSize': 'STANDARD_NC6'}
```

```
In [7]:
```

```
# For full context, report back on all the compute targets
# should now see one entry named "gpu-cluster" of type AmlCompute.
compute_targets = ws.compute_targets
```

```
for name, ct in compute_targets.items():
    print(name, ct.type, ct.provisioning_state)
```

```
Hyperdrive-01 AmlCompute Succeeded
tester321 ComputeInstance Succeeded
Hyperdrive-02 AmlCompute Succeeded
BasicCPU ComputeInstance Succeeded
Hyperdrive-03 AmlCompute Succeeded
```

AzureML Environment

Having a compute instance/cluster doesn't mean we have the right python packages installed on that compute. This is referred to as an 'environment'. We can either use YAML to specify the packages and a docker file, or we can use a curated environment. Here we sue a curated tensorflow 2.3 environment.

More at <https://docs.microsoft.com/en-us/azure/machine-learning/how-to-use-environments>

Curated environments listed at: <https://docs.microsoft.com/en-us/azure/machine-learning/resource-curated-environments>

```
In [8]:
```

```
tf_env = Environment.get(ws, name='AzureML-TensorFlow-2.1-GPU')
```

Training Script

A parameterized training script will be created in the `experiment_folder` with parameters for experimenting on varieties of the `latent_space` and `layer_sizes` arguments.

The script downloads the datasets from the workspace and trains against them, each instance with differing `latent_space` and `layer_sizes`.

Running the cell below will generate this training script.

```
In [9]:
```

```
# create a folder for the scripts
experiment_folder = './embedding-wants-autoenc-hyperdrive'
os.makedirs(experiment_folder, exist_ok=True)

print('The script folder has been created.')

# set up the experiment in AzureML
experiment = Experiment(workspace = ws,
                        name      = 'embedding-wants-autoenc-hyperdrive')

print("Experiment name : " + experiment.name)
```

```
The script folder has been created.
```

```
Experiment name : embedding-wants-autoenc-hyperdrive
```

```
In [10]:
```

```
%%writefile $experiment_folder/autoencoder_training.py

# Above Line writes this code chunk as a script, it will be used by the compute for each run of the experiment
# It gets written to the 'experiment_folder' in the cloud

# import basics
import os
import argparse
import time

# import tools for working with the data
import numpy as np
import pandas as pd
```

```

# import tensorflow's keras
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.layers import Dense, Input, Layer
from tensorflow.keras.callbacks import EarlyStopping

# import AzureML
import azureml.core
from azureml.core import Workspace, Experiment, Environment, ScriptRunConfig, Dataset, Run

print("Tensorflow Version:", tf.__version__)
print("Is tensorflow executing eagerly?", tf.executing_eagerly())
print("Is tensorflow using GPU?", tf.test.is_built_with_gpu_support())
print("Is tensorflow using Cuda?", tf.test.is_built_with_cuda())

# We get the latent_space and layer sizes for each individual experiment from the file args of this python file
parser = argparse.ArgumentParser()

# the datafolder to access the dataset is the most fiddly argument, involving some magic and numerous problems
# We pass to this script a 'dataset.as_mount()' object, via ScriptRunConfig, but this script receives a string
# That string would be presented to this script as a string variable,
# eg ${AZUREML_DATAREFERENCE_GUID}, which refers to a temp folder on the compute instance
# https://azure.github.io/azureml-examples/docs/cheatsheet/data/

parser.add_argument('--datafolder', type=str, dest='data_folder')
parser.add_argument('--latentSpace', type=int, dest='latent_space')
parser.add_argument('--layerSizes', type=int, dest='layer_sizes')
parser.add_argument('--batchSize', type=int, dest='batch_size')

# get the args
args = parser.parse_args()

# convert args to variables in this script
data_folder = args.data_folder
latent_space = args.latent_space
layer_sizes_ = args.layer_sizes
batch_size = args.batch_size
input_shape = 768

# swap layer_size ints for list of layer sizes
if layer_sizes_ == 1:
    layer_sizes = [512, 256, 128, 64]

elif layer_sizes_ == 2:
    layer_sizes = [512, 256, 128]

assert type(layer_sizes)==list, "The arg --layersizes must be 1 or 2"
# Get the experiment's currently available 'run' context
# A run represents a single trial of an experiment.
# Runs are used to monitor the asynchronous execution of a trial, log metrics and store outputs

# workspace and run context
run = Run.get_context()
ws = run.experiment.workspace # Workspace.from_config()

print("Workspace:", ws.name)

# ALTERNATIVE METHOD OF GETTING DATA
# Similar approach to using a notebook in AzureML studio

# dataset
# run_dataset = Dataset.get_by_name(ws, 'embedding-wants-dataset')
# mount_context = run_dataset.mount() # mount_point='tmp/data'

# start the mount context
# mount_context.start()

# Get the mount point
# data_folder_mp = mount_context.mount_point
# print("Mount point:", data_folder)

# List the files in the mount point
# files = os.listdir(data_folder_mp)
# print("Files available:", files)

# get paths to each file
train_path = os.path.join(data_folder, 'embeddings_sentence_train.npy')
valid_path = os.path.join(data_folder, 'embeddings_sentence_valid.npy')
testi_path = os.path.join(data_folder, 'embeddings_sentence_testi.npy')

print("train_path: ", train_path)

# Load train, valid and test numpy arrays
embeddings_train_npy = np.load(file=train_path)
embeddings_valid_npy = np.load(file=valid_path)

```

```

embeddings_testi_npy = np.load(file=testi_path)

print("train shape: ", embeddings_train_npy.shape)

# end the count context, only required if we used mount_context.start()
# mount_context.stop()

## SAMPLING FUNCTION FOR VAE #####
# Code adapted from https://keras.io/examples/generative/vae/

class Sampling(Layer):
    def __init__(self, name, **kwargs):
        super(Sampling, self).__init__(name=name, **kwargs)
        # self.new_variable = new_variable

    def call(self, inputs, training=False):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

    def get_config(self): # this is required for model_save
        config = super(Sampling, self).get_config()
        return config

# function to create autoencoder model based on args
def get_autoencoder(input_shape, layer_sizes, latent_space, activation='relu'):

    n_layers = len(layer_sizes)

    # input layer
    x = Input(shape=(input_shape,), name='input')

    h = x

    # encoder
    for i in range(n_layers):
        h = Dense(layer_sizes[i], activation=activation, name = 'encoder_'+str(i))(h)

    # VAE Latent space, no activation
    z_mean = Dense(latent_space, name='z_mean')(h)
    z_log_var = Dense(latent_space, name='z_log_var')(h)
    z_sample = Sampling(name='z_sample')([z_mean, z_log_var])

    # decoder, reads layer_sizes in reverse order
    h = z_sample
    for i in range(n_layers-1, -1, -1):
        h = Dense(layer_sizes[i], activation=activation, name = 'decoder_'+str(i))(h)

    # output Layer, no activation
    h = Dense(input_shape, name = 'output_layer')(h)

    # model
    model = Model(inputs = x, outputs = h)
    model.summary()

    return model

# use the function to create the model
autoenc = get_autoencoder( input_shape = input_shape,
                           layer_sizes = layer_sizes,
                           latent_space = latent_space)

# The encoder model will simply be a keras model built from the autoencoder's layers
# The encoder's output is the latent space list; [z_mean, z_log_var, z_sample]. Each with shape [batch, latent_space]
# When the model is training we focus on z_sample, but when predicting we focus on z_mean
z_mean = autoenc.get_layer(name='z_mean').output
z_log_var = autoenc.get_layer(name='z_log_var').output
z_sample = autoenc.get_layer(name='z_sample').output

# The encoder model will simply be a keras model built from the autoencoder's layers
# output is a list: [z_mean, z_log_var, z_sample]. Each with shape [batch, latent_space]
encoder = Model(inputs=autoenc.input, outputs=[z_mean, z_log_var, z_sample], name="encoder")

# compile the autoencoder
autoenc.compile(loss='mse', optimizer='adam')
earlystop = EarlyStopping(monitor='val_loss', mode='min', patience=8, verbose=1)

# get start time for training
start_time = time.perf_counter()

# train the model
history = autoenc.fit(x = embeddings_train_npy,
                       y = embeddings_train_npy,
                       batch_size=batch_size,

```

```

        epochs      = 64,
        verbose     = 2,
        callbacks   = [earlystop],
        shuffle     = True,
        validation_data=(embeddings_valid_npy, embeddings_valid_npy)
    )

# record time to train
stop_time      = time.perf_counter()
training_time = (stop_time - start_time)
run.log("train_time", training_time)
print("Total time in seconds for training: {}".format(str(training_time)))

# evaluate the model on the test set
testi_loss = autoenc.evaluate(x=embeddings_testi_npy, y=embeddings_testi_npy, verbose=0)

# Log the metric, i.e. reconstruction error on test set, in AzureML experiment
run.log("mse_test", float(testi_loss))

# files saved in the "./outputs" folder are automatically uploaded into run history
os.makedirs('./outputs/model', exist_ok=True)
run_name = "Run_latent_" + str(latent_space) + "_lyrsizes_" + str(layer_sizes_) + "_batch_" + str(batch_size)

# save the autoencoder (ie encoder+decoder) and encoder (standalone) models
autoenc.save(os.path.join('./outputs/model/', run_name+'_autoenc_model.pb'))
encoder.save(os.path.join('./outputs/model/', run_name+'_encoder_model.pb'))

# save the weights
autoenc.save_weights(os.path.join('./outputs/model/', run_name+'_autoenc_weights.h5'))
encoder.save_weights(os.path.join('./outputs/model/', run_name+'_encoder_weights.h5'))

# save the training history, so we can view the training chart. Uses pandas to_csv() function
history_pd = pd.DataFrame(history.history)
history_pd_filename = os.path.join('./outputs/model/', run_name+'_history.csv')
with open(history_pd_filename, mode='w') as f:
    history_pd.to_csv(f)

# end run
run.complete()

```

Overwriting ./embedding-wants-autoenc-hyperdrive/autoencoder_training.py

Prepare Config for a Hyperdrive Experiment

AzureML allows us to sample the hyperparams to be tested. Currently, the sampling methods supported are random sampling, grid sampling, and Bayesian sampling. With random sampling, you can use continuous hyperparameters that choose within a range of values instead of statically calling out each value to use. This reduces some of the manual work in hyperparameter tuning.

However, we simply need a grid approach. The following code chunk prepares the params to be explored and the configuration of the experiment to be conducted.

Note the latent spaces are hand picked: 30, 60 or 104 104 equates to the number of dims PCA requires to retain 80% of the variance, so we can compare PCA with Autoencoder. 30 and 60 are simply guesses, although 30 is approximately the maximum number we can use in clustering with this number of records. 500k+ points become very sparse is more than 30 dims.

In [11]:

```

# Parameter values for random sampling
params = GridParameterSampling(
    {
        '--latentspace': choice(30, 60, 100),
        '--layersizes' : choice(1, 2),
        '--batchsize'  : choice(100) # VAE's need sufficient sample size, typically 100. 64 may also work.
    }
)

# setup how each run of the model will be configured
# Note, the dataset.as_mount() command. This is crucial.
# It returns a DatasetConsumptionConfig object with the mode set to mount
# This allows the dataset to be mounted to a temporary Local path on each compute target
config = ScriptRunConfig(source_directory = experiment_folder,
                        script           = 'autoencoder_training.py',
                        arguments        = ['--datafolder', embeddings_dataset.as_mount()], #path_on_compute='tmp\data'
                        compute_target   = training_cluster,
                        environment     = tf_env
)

```

In [12]:

```

# Configure hyperdrive settings
# this is for the group of runs which comprises the hyperdrive experiment

# set hyperparam search policy for early termination
## COMMENTED OUT, NOT USED
# early_termination_policy = BanditPolicy(slack_factor=0.15, evaluation_interval=1, delay_evaluation=10)

```

```

hyperdrive = HyperDriveConfig(run_config      = config,
                             hyperparameter_sampling= params,
                             policy                = None, #early_termination_policy
                             primary_metric_name   = 'mse',
                             primary_metric_goal   = PrimaryMetricGoal.MINIMIZE,
                             max_total_runs        = 12,
                             max_concurrent_runs   = 4)

print("The hyperdrive is ready to run!")

```

The hyperdrive is ready to run!

AzureML Experiment

The final task is to execute the hyperdrive config on the experiment which we have already instantiated

In [13]:

```

# Execute the experiment with the hyperdrive config
hyperdrive_run = experiment.submit(config=hyperdrive)

# Show the status
RunDetails(hyperdrive_run).show()
hyperdrive_run.wait_for_completion()

```

```

Out[13]: {'runId': 'HD_3fa3599a-fc32-4711-9d65-c1920571bb49',
           'target': 'Hyperdrive-01',
           'status': 'Completed',
           'startTimeUtc': '2021-07-07T13:19:36.581871Z',
           'endTimeUtc': '2021-07-07T14:41:40.814406Z',
           'properties': {'primary_metric_config': '{"name": "mse", "goal": "minimize"}',
                          'resume_from': 'null',
                          'runTemplate': 'HyperDrive',
                          'azureml.runsource': 'hyperdrive',
                          'platform': 'AML',
                          'ContentSnapshotId': 'dbdd21e9-6e92-466f-b085-8c6269c04967'},
           'inputDatasets': [],
           'outputDatasets': [],
           'logFiles': {'azureml-logs/hyperdrive.txt': 'https://autoencwstoraged8ea79dd7.blob.core.windows.net/azureml/ExperimentRun/dciHD_3fa3599a-fc32-4711-9d65-c1920571bb49/azureml-logs/hyperdrive.txt?sv=2019-02-02&sr=b&sig=MKY6MVu15WLcBY%2FYzydvBj1GPiu9KUdq6NWKdwJaF6g%3D&t=2021-07-07T14%3A31%3A55Z&se=2021-07-07T22%3A41%3A55Z&sp=r'},
           'submittedBy': 'Morris Corfield'}

```

Best run

Normally we would simply want the best run. But in this case, we know models with larger latent spaces will have lower reconstruction errors. We want as few latent dimensions as possible and we would like the autoencoder to give lower reconstruction errors for the same number of dimensions as PCA. Otherwise, we may simply use PCA.

So, we need to study reconstruction errors of each autoencoder latent space with PCA applying the same number of dimensions. If the autoencoder is better with 60 dims than PCA, then we will take the autoencoder. If the loss with 30 dims is comparable with the loss with 60 dims, then we will take 30.

Summary of Runs

We want to view a summary of the results from our most recent experiment. A summary looking like the one available in the AzureML studio. Curiously, AzureML has no function for presenting the summary available in the Studio, so we have to code it:

In [14]:

```

# get the experiment by name
experiment = Experiment(workspace=ws, name='embedding-wants-autoenc-hyperdrive')

```

In [16]:

```

# function to get the experiment results, same as summary visible in the AzureML studio
def get_last_experiment_results(experiment, sort_by_metric, minimise=True):

    # azureML stores each experiment as a 'run' with multiple child runs
    # this is a little confusing, as only those child runs are real runs; executed to carry out an experiment
    # We need the final batch of runs, azureML stores the list of experiment runs in reverse chronological order
    # so the first in the list is the most recent, the one we want
    final_runs, *_ = experiment.get_runs(include_children=False)

    # All the real runs are children to the experiment's 'run'
    child_runs = final_runs.get_children()

    # We will require a function to convert a list of arguments into a dict of arguments
    def ConvertListToDict(list_instance):
        iterable = iter(list_instance)
        dictionary = dict(zip(iterable, iterable))
        return dictionary

```

```

# now we can cycle thru the child runs and pick out the useful summary information
runs_list = []
for child in child_runs:
    # get the run's metrics (eg mse) and details
    # details has many parameters, we will filter down to what need in the following code
    metrics_dict = child.get_metrics()
    details_dict = child.get_details()

    # get the run ID from the details
    runId_dict = {'runId': details_dict['runId']}

    # get the arguments from the details
    # these arguments pare those passed to the run's script, eg layer_size, batch_size, etc
    argumts_list = details_dict['runDefinition']['arguments']
    argumts_dict = ConvertListToDict(argumts_list)

    # add the run's metrics and arguments to each runID
    # result is one dict for the run which has ID, metrics and arguments
    runId_dict.update(metrics_dict)
    runId_dict.update(argumts_dict)

    # append that this run's dict to a list
    runs_list.append(runId_dict.copy())

# our data is now a list of dicts, so can be converted to pandas
runs_list = pd.DataFrame(runs_list)

# don't need the datafolder column
if '--datafolder' in runs_list.columns:
    runs_list = runs_list.drop(['--datafolder'], axis=1)

# sort by the 'sort_by_metric'
runs_list = runs_list.sort_values(sort_by_metric, ascending=minimise)

# rename columns, removing '--' from names
runs_list.columns = [column.replace("--", "") for column in runs_list.columns]

# print to screen
return runs_list

```

In [17]:

```

runs_list = get_last_experiment_results(experiment      = experiment,
                                         sort_by_metric = 'mse_test',
                                         minimise       = True)
runs_list

```

Out[17]:

	runId	train_time	mse_test	latentspace	layersizes	batchsize
0	HD_3fa3599a-fc32-4711-9d65-c1920571bb49_5	1386.434425	0.163011	100	2	100
1	HD_3fa3599a-fc32-4711-9d65-c1920571bb49_4	936.506901	0.170917	60	2	100
3	HD_3fa3599a-fc32-4711-9d65-c1920571bb49_1	1376.063717	0.228218	60	1	100
2	HD_3fa3599a-fc32-4711-9d65-c1920571bb49_2	1672.535477	0.229278	100	1	100
5	HD_3fa3599a-fc32-4711-9d65-c1920571bb49_0	1225.270789	0.232384	30	1	100
4	HD_3fa3599a-fc32-4711-9d65-c1920571bb49_3	812.495413	0.233903	30	2	100

Select the Best Run

The best run has a latent_space of 100 dims, which was included as a comparison with the PCA compression which had a similar number of dims for retention of 80% of variance.

However, we would like to compress the data more than that and see that 60 dims presents only a small increase in reconstruction error from 0.163 to 0.171. Whereas further compression to 30 dims demands we accept much greater reconstruction error of 0.232

So, a latent space of 60 dims looks like a good compromise between compression and reconstruction error. This could be refined, of course, using many more runs to plot the relationship between latent space and reconstruction errors. However, we are short of time.

So, the best model with such a latent space has a batch size of 60 and a layer_size of '2', which represents layers sizes like this [512, 256, 128]

This is our best run, now we need to download that autoencoder and associated encoder, so we can properly compare it with PCA.

In [18]:

```

# normally we'd get the best run using this command...
# best_run = hyperdrive_run.get_best_run_by_primary_metric()
# But on this occasion we want the run with the below params:

best_run_id = runs_list.query("(latentspace=='60') and (layersizes=='2') and (batchsize=='100')")
best_run_id = best_run_id['runId'].iloc[0]

```

In [19]:

```
## Download the Best Model

# If training the model took many hours then we would want to register it.
# model = best_run.register_model(model_name='tf-dnn-mnist', model_path='outputs/model')
best_run = Run(experiment=experiment, run_id=best_run_id)

# download the autoencoder and encoder
output_file_path=''
for filename in best_run.get_file_names():
    if 'autoenc' in filename:
        output_file_path = os.path.join(os.getcwd(), 'Autoencoder')
    elif 'encoder' in filename:
        output_file_path = os.path.join(os.getcwd(), 'Encoder')
    elif 'history' in filename:
        output_file_path = os.path.join(os.getcwd(), 'History')
    else:
        pass

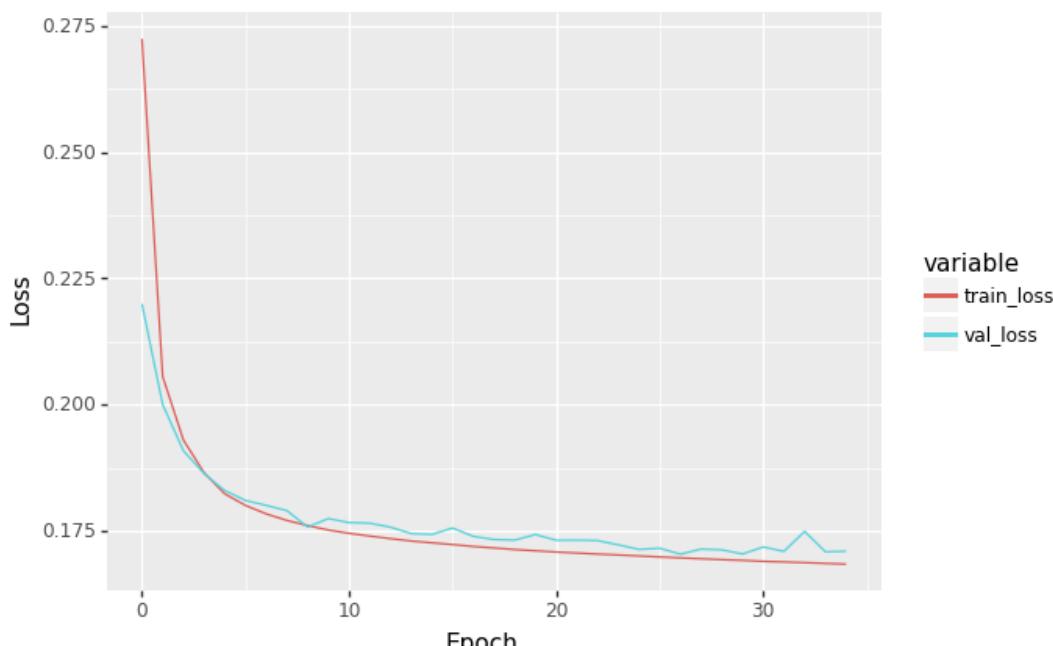
    if output_file_path != '':
        print('Downloading from {} to {} ...'.format(filename, output_file_path))
        best_run.download_file(name=filename, output_file_path=output_file_path)

output_file_path =''
```

```
Downloading from outputs/model/Run_latent_60_lyrsizes_2_batch_100_autoenc_model.pb/saved_model.pb to /home/oliver/Documents/Deep/Data/Autoencoder ...
Downloading from outputs/model/Run_latent_60_lyrsizes_2_batch_100_autoenc_model.pb/variables/variables.data-00000-of-00001 to /home/oliver/Documents/Deep/Data/Autoencoder ...
Downloading from outputs/model/Run_latent_60_lyrsizes_2_batch_100_autoenc_model.pb/variables/variables.index to /home/oliver/Documents/Deep/Data/Autoencoder ...
Downloading from outputs/model/Run_latent_60_lyrsizes_2_batch_100_autoenc_weights.h5 to /home/oliver/Documents/Deep/Data/Autoencoder ...
Downloading from outputs/model/Run_latent_60_lyrsizes_2_batch_100_encoder_model.pb/saved_model.pb to /home/oliver/Documents/Deep/Data/Encoder ...
Downloading from outputs/model/Run_latent_60_lyrsizes_2_batch_100_encoder_model.pb/variables/variables.data-00000-of-00001 to /home/oliver/Documents/Deep/Data/Encoder ...
Downloading from outputs/model/Run_latent_60_lyrsizes_2_batch_100_encoder_model.pb/variables/variables.index to /home/oliver/Documents/Deep/Data/Encoder ...
Downloading from outputs/model/Run_latent_60_lyrsizes_2_batch_100_encoder_weights.h5 to /home/oliver/Documents/Deep/Data/Encoder ...
Downloading from outputs/model/Run_latent_60_lyrsizes_2_batch_100_history.csv to /home/oliver/Documents/Deep/Data/History ...
```

In [20]:

```
# Lets see the training history of the best model
# wrangle
history = pd.read_csv(os.path.join(os.getcwd(), 'History'))
history.columns = ['epoch', 'train_loss', 'val_loss']
history = pd.melt(history, id_vars=['epoch'], value_vars=['train_loss', 'val_loss'])
# plot
(
    p9.ggplot(history)
    + p9.aes(x='epoch', y='value', color='variable', group='variable')
    + p9.geom_line()
    + p9.labs(x='Epoch', y='Loss')
)
```



Out[20]: <ggplot: (8764636783074)>

Compare Encoder with PCA

We need to show that all the effort of training an autoencoder delivers a compression tool better than PCA, which is much simpler.

In [21]:

```
# Lets load the data again...
embeddings_train = np.load(os.path.join(path, 'embeddings_sentence_train.npy'))
embeddings_testi = np.load(os.path.join(path, 'embeddings_sentence_testi.npy'))
embeddings_valid = np.load(os.path.join(path, 'embeddings_sentence_valid.npy'))
```

Normally we'd download the keras model and execute it, simple. But the below code is causing errors shape errors. This may be because Tensorflow in the AzureML environment is v2.1, whereas the local PC is running v2.4.

The alternative approach is to create an identical instance of the model and load the weights from the training.

In []:

```
##### THIS CODE FAILS. SHAPE ERRORS #####
# to compare the autoencoder with PCA, get the autoencoder
autoenc_path = os.path.join(os.getcwd(), 'Autoencoder')

# We are using a custom Layer, Sampling, so need create that Layer in our current context
class Sampling(Layer):
    def __init__(self, name, **kwargs):
        super(Sampling, self).__init__(name=name, **kwargs)
    def call(self, inputs, training=False):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon
    def get_config(self): # this is required for model_save
        config = super(Sampling, self).get_config()
        return config

# now we can load the model, telling keras about our custom layer
autoenc_model = tf.keras.models.load_model(filepath = autoenc_path,
                                            custom_objects= {'Sampling': Sampling},
                                            compile = False)
```

In [22]:

```
##### ALTERNATIVE. CREATE INSTANCE OF MODEL AND APPLY SAVED WEIGHTS #####
class Sampling(Layer):
    def __init__(self, name, **kwargs):
        super(Sampling, self).__init__(name=name, **kwargs)
    def call(self, inputs, training=False):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon
    def get_config(self):
        config = super(Sampling, self).get_config()
        return config

def get_autoencoder(input_shape, layer_sizes, latent_space, activation='relu'):
    n_layers = len(layer_sizes)
    x = Input(shape=(input_shape,), name='input')
    h = x
    for i in range(n_layers):
        h = Dense(layer_sizes[i], activation=activation, name = 'encoder_'+str(i))(h)
    z_mean = Dense(latent_space, name='z_mean')(h)
    z_log_var = Dense(latent_space, name='z_log_var')(h)
    z_sample = Sampling(name='z_sample')([z_mean, z_log_var])
    h = z_sample
    for i in range(n_layers-1, -1, -1):
        h = Dense(layer_sizes[i], activation=activation, name = 'decoder_'+str(i))(h)
    h = Dense(input_shape, name = 'output_layer')(h)
    model = Model(inputs = x, outputs = h)
    model.summary()
    return model

# create instance of model, using optimal params
autoenc = get_autoencoder( input_shape = 768,
                           layer_sizes = [512,256,128],
                           latent_space = 60)

# Load weights:
# note you may need to call .expect_partial() at end of this line
# if TF throws warnings that we are only using the weights req'd for inference, ie many unused weights
autoenc.load_weights(os.path.join(path, 'Autoencoder', 'Run_latent_60_lyrsizes_2_batch_100_autoenc_weights.h5'))

# create encoder from autoencoder layers
```

```

z_mean    = autoenc.get_layer(name='z_mean').output
z_log_var = autoenc.get_layer(name='z_log_var').output
z_sample  = autoenc.get_layer(name='z_sample').output

# The encoder model will simply be a keras model built from the autoencoder's layers
# output is a list: [z_mean, z_log_var, z_sample]. Each with shape [batch, latent_space]
encoder = Model(inputs=autoenc.input, outputs=[z_mean, z_log_var, z_sample], name="encoder")

```

WARNING:tensorflow:AutoGraph could not transform <bound method Sampling.call of <__main__.Sampling object at 0x7f8b580aaa30>> and will run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERTBOSITY=10`) and attach the full output.
Cause: module 'gast' has no attribute 'Index'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
WARNING: AutoGraph could not transform <bound method Sampling.call of <__main__.Sampling object at 0x7f8b580aaa30>> and will run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERTBOSITY=10`) and attach the full output.
Cause: module 'gast' has no attribute 'Index'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input (InputLayer)	[None, 768]	0	
encoder_0 (Dense)	(None, 512)	393728	input[0][0]
encoder_1 (Dense)	(None, 256)	131328	encoder_0[0][0]
encoder_2 (Dense)	(None, 128)	32896	encoder_1[0][0]
z_mean (Dense)	(None, 60)	7740	encoder_2[0][0]
z_log_var (Dense)	(None, 60)	7740	encoder_2[0][0]
z_sample (Sampling)	(None, 60)	0	z_mean[0][0] z_log_var[0][0]
decoder_2 (Dense)	(None, 128)	7808	z_sample[0][0]
decoder_1 (Dense)	(None, 256)	33024	decoder_2[0][0]
decoder_0 (Dense)	(None, 512)	131584	decoder_1[0][0]
output_layer (Dense)	(None, 768)	393984	decoder_0[0][0]

Total params: 1,139,832
Trainable params: 1,139,832
Non-trainable params: 0

In [23]:

```

# use the autoencoder model to get reconstructions of the test data
autoenc_recons_testi = autoenc.predict(embeddings_testi)

# get reconstruction error on test data
autoenc_loss_testi = np.mean((embeddings_testi - autoenc_recons_testi)**2)

```

In [24]:

```

# We can compare the mean square error

# Fit PCA with same number of dims as used in the autoencoder Latent-space, ie same compression ratio as the autoencoder
# use the training set, because we trained the autoencoder on the training set
latent_space = 60

pca_n = PCA(n_components=latent_space)
pca_n.fit(embeddings_train)

# Get PCA transforms for n=latent_space
# Use the validation dataset because we calculated autoencoder loss on the validation set
pca_n_embeddings_testi = pca_n.fit_transform(embeddings_testi)

# Reconstruct the 768dim validation embeddings from the n_dim PCA transforms
pca_n_recons_testi = pca_n.inverse_transform(pca_n_embeddings_testi)

# Calculate Mean Squared Error (MSE) for PCA reconstructions of validation data
pca_n_loss_testi = np.mean((embeddings_testi - pca_n_recons_testi)**2)

print('For dims = ',latent_space)
print('MSE for Autoencoder reconstructions : %.3f' % (autoenc_loss_testi))
print('MSE for PCA reconstructions      : %.3f' % (pca_n_loss_testi))

For dims = 60
MSE for Autoencoder reconstructions : 0.171
MSE for PCA reconstructions      : 0.245

```

The autoencoder is substantially better than PCA on the validation dataset. Autoencoders are well known to be better at compression of non-linear relationships between dimensions. Clearly the Albert embeddings have such relationships ready to be exploited

UMAP

We have compressed the data to 60 dimensions, yet this is still too many for HDBSCAN, which is the preferred clustering method. Hierarchical clustering in 60 dimensions take an enormous amount of computing power. The task will be more achievable in half as many dims, ie 30.

UMAP is a useful tool here. UMAP is NOT the same type of compression as PCA or autoencoding. At this stage we are interested in maintaining relative distance between points, ie in clustering. It is acceptable to sacrifice the accuracy of reconstruction so long as the potential for clustering is maintained.

We could have applied UMAP without any prior compression by PCA or autoencoder. However, with 768 dimensions UMAP would have been impossibly slow, so the first compression step was necessary.

Note, T-SNE is alternative approach to UMAP.

```
In [25]: # The encoder model returns a List: [z_mean, z_log_var, z_sample]. Each item has shape [batch, latent_space]
# We need item 0, z_mean, for the predictions.
embeddings_train_enc = encoder.predict(embeddings_train)[0]

# save results
np.save(file=os.path.join(path, 'embeddings_train_enc.npy'), arr=embeddings_train_enc)
```

Further Dimension Reduction

The favoured clustering method for this data is HDBSCAN because it discovers the number of clusters which the data presents, whereas other methods require us to impose a number of clusters up front. However, HDBSCAN is very slow when attempted on 300k rows of data each row with 60dimensions. Other clustering methods such GMM (Gaussian Mixture Model) may suit this kind of data and work quickly in 60dims, but demand we impose our assumption of cluster quantity on the data.

So, we will further reduce the dimensions to just 30. This can be done using UMAP, which will retain the information required for clustering and discard other information in order to halve the dimensionality of the data.

Explore Clustering Parameters

Should UMAP be applied with cosine distances or manhattan? How do the clusters appear when charted in 2dimensions, are they convincing? Are any clusters found or is it largely one homogenous cloud?

We need to investigate these options before progressing.

```
In [26]: # get the data
embeddings_train_enc = np.load(os.path.join(path, 'embeddings_train_enc.npy'))

# List the dimensions and metrics we will consider
dims_list = [2, 30]
metric_list = ['cosine', 'manhattan']
# get combinations of above two lists
dims_metric_list = pd.DataFrame(list(product(dims_list, metric_list)), columns=['dims', 'metric'])

# subsample the data
sample_size = 50000
indexes = np.arange(embeddings_train_enc.shape[0])
print("Number of rows of data: ", len(indexes))
np.random.shuffle(indexes)

for row in dims_metric_list.itertuples():

    start_time = time.time()
    print("Commencing UMAP with dims =", int(row.dims), ". Metric =", row.metric)

    # project data down to dims
    umap_reducer = umap.UMAP(n_components = row.dims, metric = row.metric,
                            random_state = 0, n_neighbors = 30, min_dist = float(0.00))
    embeddings_train_enc_umap = umap_reducer.fit_transform(embeddings_train_enc)

    print("Commencing clustering")
    # create clusterer in proportion to size of subset, then cluster ALL the data (not just sample)
    clusterer = hdbscan.HDBSCAN(metric = 'manhattan', min_cluster_size= 3000, min_samples = 1000)
    clusterer.fit(embeddings_train_enc_umap)

    print("Sampling data for plots")
    # get sample
    sample_data = embeddings_train_enc_umap[indexes[:sample_size],:]

    # grab the 2D data, we'll need it for plotting
    # First time around the loop we create the numpy object from scratch
    if row.dims == dims_metric_list['dims'][0] and row.metric == dims_metric_list['metric'][0]:
        embeddings_train_enc_umap_samples_2D = sample_data
```

```

# assemble the subset; 2D data for plotting, cluster for dims, dims
# Note, we mix the data for 2dims (which can be plotted) with the clusters for multiple dims
# Also, can't use metric string in np array, so use index instead
metric_index = metric_list.index(row.metric)

subset = np.hstack((np.zeros( (sample_size, 1 )+row.dims, # dims
                             np.zeros( (sample_size, 1 )+metric_index, # metric index
                             clusterer.labels_[indexes[:sample_size],np.newaxis], # cluster assignment
                             embeddings_train_enc_umap_samples_2D))) # embeddings

# append the data. Special rule for first iteration
if row.dims==dims_metric_list['dims'][0] and row.metric==dims_metric_list['metric'][0] :
    embeddings_train_enc_umap_samples = subset
else:
    embeddings_train_enc_umap_samples = np.vstack((embeddings_train_enc_umap_samples,
                                                    subset))

# we should keep a record of the umap data too, took a long time to create
embeddings_train_enc_umap_all = np.hstack(( np.zeros( (embeddings_train_enc_umap.shape[0], 1 )+row.dims, # dims
                                                       np.zeros( (embeddings_train_enc_umap.shape[0], 1 )+metric_index, # metric index
                                                       clusterer.labels_[:,np.newaxis], # cluster ass
                                                       embeddings_train_enc_umap )) # embeddings

# save the file of all embeddings with their cluster assignments
np.save(file= os.path.join(path, 'embeddings_train_enc_umap_dims'+str(int(row.dims))+'_'+row.metric+'.npy'),
        arr = embeddings_train_enc_umap_all)

# Report time for loop
print("Elapsed time for Dims = ", str(int(row.dims)),
      ", Metric = ", row.metric,
      " : ", str((time.time()-start_time)/60) )

# save the collection of samples
np.save(file= os.path.join(path, 'embeddings_train_enc_umap_samples_2D.npy'),
        arr = embeddings_train_enc_umap_samples)

```

Number of rows of data: 369930
Commencing UMAP with dims = 2 . Metric = cosine
Commencing clustering
Sampling data for plots
Elapsed time for Dims = 2 , Metric = cosine : 11.629086530208587
Commencing UMAP with dims = 2 . Metric = manhattan
Commencing clustering
Sampling data for plots
Elapsed time for Dims = 2 , Metric = manhattan : 10.627359143892924
Commencing UMAP with dims = 30 . Metric = cosine
Commencing clustering
Sampling data for plots
Elapsed time for Dims = 30 , Metric = cosine : 18.48861695130666
Commencing UMAP with dims = 30 . Metric = manhattan
Commencing clustering
Sampling data for plots
Elapsed time for Dims = 30 , Metric = manhattan : 22.459272193908692

In [27]:

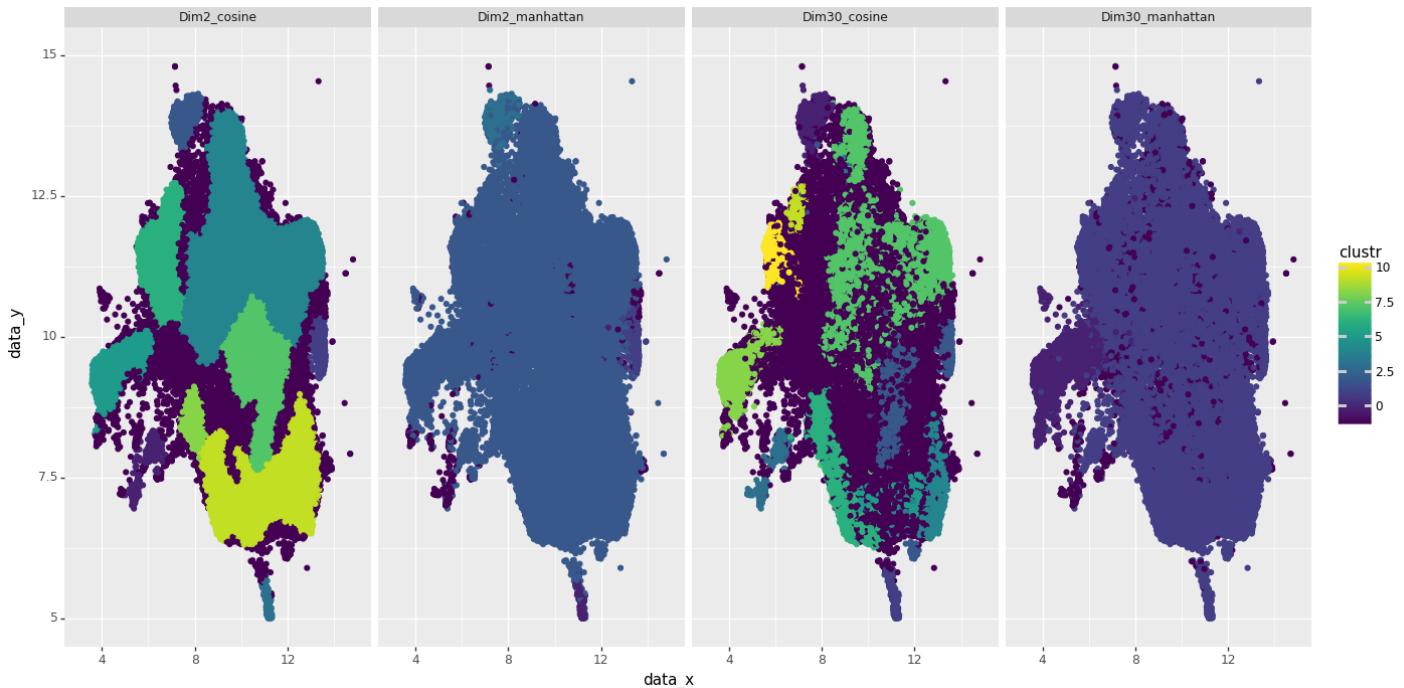
```

# Let's view those clusters in 2d
clusters_umap2D = pd.DataFrame({'dims_metric':[ 'Dim'+str(int(dim))+'_'+str(metric_list[int(metric)]) for dim, metric in embeddings_train_enc_umap_samples[:,0:2] ],
                                    'clustr':embeddings_train_enc_umap_samples[:,2],
                                    'data_x':embeddings_train_enc_umap_samples[:,3],
                                    'data_y':embeddings_train_enc_umap_samples[:,4]})

# plot the data
(
    p9.ggplot(clusters_umap2D)
    + p9.aes(x='data_x', y='data_y')
    + p9.geom_point(p9.aes(color='clustr'))
    + p9.facet_grid('. ~ dims_metric')
    + p9.scales.xlim(3,15)
    + p9.scales.ylim(5,15)
    + p9.theme.figure_size=(16, 8))
)

```

Index._and_. operating as a set operation is deprecated, in the future this will be a logical operation matching Series._and_. Use index.intersection(other) instead
Index._and_. operating as a set operation is deprecated, in the future this will be a logical operation matching Series._and_. Use index.intersection(other) instead
geom_point : Removed 1524 rows containing missing values.



Out[27]: <ggplot: (8764038712802)>

Use AzureML to Optimise HDBSCAN Clustering

The above results show that UMAP with cosine distances and data in 30dims leads to convincing clusters. These have been plotted in just 2dims and the global structure is clearly retained. These are clusters worth optimising.

The parameters for optimising with HDBSCAN are min_cluster_size and min_samples. We will use AzureML to carry out a number of experiments and see how these parameters affect the clustering results.

In [28]:

```
# We will now use UMAP to further reduce the dimensions
# UMAP is used in preference to t-SNE following the arguments presented in this blog:
# https://towardsdatascience.com/tsne-vs-umap-global-structure-4d8045acba17

# We will aim to halve the number of dimensions used by the encoder
# known as n_components

# declare umap object, manhattan distance preferred for high dimensional space
umap_reducer = umap.UMAP(random_state= 0,
                           metric      = 'cosine',
                           n_components= 30,
                           n_neighbors = 30,
                           min_dist     = float(0.00) )

# fit umap transform, timing the process as we go
start = time.time()
print('Starting umap process')
embeddings_train_enc_umap = umap_reducer.fit_transform(embeddings_train_enc)
print('Time taken to prepare umap = %.2f mins' % ((time.time()-start)/60) )

# save the UMAP data to the AzureML folder,
# note reduced to 30 dims, not 768 dims, much smaller file than original embeddings.
np.save(file = os.path.join(path, 'AzureML', 'enc_umap', 'embeddings_train_un_enc_umap_30.npy'),
        arr = embeddings_train_enc_umap)
```

Starting umap process
Time taken to prepare umap = 16.60 mins

In [29]:

```
# we'll also upload this to a new dataset in AzureML

# upload data to the default AzureML datastore
# Does not upload single file, must be entire folder
ws = Workspace.get( name      = 'Autoenc_ws',
                    subscription_id = 'GUID',
                    resource_group  = 'AzureML_PAYG')

datastore = ws.get_default_datastore()
```

In [30]:

```
# upload
datastore.upload(src_dir      = os.path.join(path, 'AzureML', 'enc_umap'),
                 target_path = 'embedding_wants_enc_umap',
                 overwrite   = True)
```

```
Uploading an estimated of 1 files
Uploading /home/oliver/Documents/Deep/Data/AzureML/enc_umap/embeddings_train_un_enc_umap_30.npy
Uploaded /home/oliver/Documents/Deep/Data/AzureML/enc_umap/embeddings_train_un_enc_umap_30.npy, 1 files out of an estimated total of 1
Uploaded 1 files
```

```
Out[30]: $AZUREML_DATAREFERENCE_5e923bf222b84377988ae97ba2989f18
```

```
In [31]: # create a FileDataset around the directory of the datastore,
# use wildcard to select all (train, test, valid) numpy files into the new dataset
embeddings_enc_umap_dataset = Dataset.File.from_files(path=(datastore, 'embedding_wants_enc_umap'))
```

```
# register the data
dataset_registered = False
```

```
try:
    # can we get the registered dataset by its name?
    temp = Dataset.get_by_name(workspace = ws, name = 'embedding-wants-enc_umap')
    dataset_registered = True
except:
    print("The dataset 'embedding-wants-enc-umap' is not registered in workspace yet.")
```

```
if not dataset_registered:
    # Register the dataset, with a name
    # Use the register() method to register datasets to your workspace so they can be shared with others,
    # reused across various experiments, and referred to by name in your training script.
    embeddings_enc_umap_dataset = embeddings_enc_umap_dataset.register(workspace = ws,
                                                                      name = 'embedding-wants-enc-umap',
                                                                      description = 'clustering dataset',
                                                                      tags = {'format':'npy'},
                                                                      create_new_version= True)
```

```
# confirm the datasets we have now registered
```

```
print("Datasets Versions:")
for dataset_name in list(ws.datasets.keys()):
    dataset = Dataset.get_by_name(ws, dataset_name)
    print("\t", dataset.name, 'version', dataset.version)
```

```
The dataset 'embedding-wants-enc-umap' is not registered in workspace yet.
```

```
Datasets Versions:
embedding-wants-enc-umap version 1
embedding-wants-dataset version 1
```

No we have reduced the data down to a manageable number of dimensions we can consider clustering. We'll attempt the clustering using the entire training dataset with a selection of HDBSCAN hyperparams and see which scheme clusters the largest proportion of embeddings and the most reasonable size of clusters.

Clustering Environment

The clustering task will use the HDBSCAN package, this has a number of parameters, not least:

- min_cluster_size
- min_samples

Varying these parameters will affect the clusters which HDBSCAN finds. We'll run a number of experiments, (16 in fact) to explore the solution space. This is best done in AzureML, so runs can be executed in parallel, as we did with the autoencoder.

Unfortunately, HDBSCAN is not part of the standard AzureML packages available, so we will have to create an 'environment' for the experiment.

Instructions for working with AzureML environments can be found at: <https://azure.github.io/azureml-examples/docs/cheatsheet/environment/>

```
In [32]: #We need to see what environments are already registered to our workspace
list(ws.environments.keys())
```

```
Out[32]: ['Clustering',
          'AzureML-VowpalWabbit-8.8.0',
          'AzureML-PyTorch-1.3-CPU',
          'AzureML-Minimal',
          'AzureML-Tutorial',
          'AzureML-TensorFlow-2.3-CPU',
          'AzureML-Triton',
          'AzureML-PyTorch-1.6-CPU',
          'AzureML-PyTorch-1.6-GPU',
          'AzureML-TensorFlow-2.3-GPU',
          'AzureML-DeepSpeed-0.3-GPU',
          'AzureML-tensorflow-2.4-ubuntu18.04-py37-cuda11-gpu',
          'AzureML-tensorflow-1.15-ubuntu18.04-py37-cpu-inference',
          'AzureML-sklearn-0.24-ubuntu18.04-py37-cuda11-gpu',
          'AzureML-lightgbm-3.2-ubuntu18.04-py37-cpu',
          'AzureML-tensorflow-2.4-ubuntu18.04-py37-cpu-inference',
          'AzureML-tensorflow-2.4-ubuntu18.04-py37-cuda11.0.3-gpu-inference',
```

```
'AzureML-pytorch-1.7-ubuntu18.04-py37-cpu-inference',
'AzureML-pytorch-1.7-ubuntu18.04-py37-cuda11-gpu',
'AzureML-sklearn-0.24.1-ubuntu18.04-py37-cpu-inference',
'AzureML-minimal-ubuntu18.04-py37-cpu-inference',
'AzureML-onnxruntime-1.6-ubuntu18.04-py37-cpu-inference',
'AzureML-xgboost-0.9-ubuntu18.04-py37-cpu-inference',
'AzureML-pytorch-1.6-ubuntu18.04-py37-cpu-inference',
'AzureML-mlflow-ubuntu18.04-py37-cpu-inference',
'AzureML-tritonserver-21.02-py38-inference']
```

In [33]:

```
# Lets grab the workspace's standard 'Tutorial' environment
envs = Environment.list(workspace=ws)

for env in envs:
    if env == "AzureML-Tutorial":
        env_tutorial = envs[env]

clustering_env = Environment(name="Clustering")

# Lets adapt the AzureML-Tutorial environment with the packages required for our clustering script
clustering_env.python.conda_dependencies = env_tutorial.python.conda_dependencies
clustering_env_dep = clustering_env.python.conda_dependencies

clustering_env_dep.add_conda_package("hdbscan")
clustering_env_dep.add_conda_package("umap-learn")

# View the packages yaml
#clustering_conda_dep.serialize_to_string()
print("Environment name ", clustering_env.name)
print("packages", clustering_env.python.conda_dependencies.serialize_to_string())
```

```
Environment name Clustering
packages channels:
- anaconda
- conda-forge
dependencies:
- python=3.6.2
- pip:
  - azureml-core==1.31.0
  - azureml-defaults==1.31.0
  - azureml-telemetry==1.31.0
  - azureml-train-restclients-hyperdrive==1.31.0
  - azureml-train-core==1.31.0
  - azureml-widgets==1.31.0
  - azureml-pipeline-core==1.31.0
  - azureml-pipeline-steps==1.31.0
  - azureml-opendatasets==1.31.0
  - azureml-automl-core==1.31.0
  - azureml-automl-runtime==1.31.0
  - azureml-train-automl-client==1.31.0
  - azureml-train-automl-runtime==1.31.0
  - azureml-train-automl==1.31.0
  - azureml-train==1.31.0
  - azureml-sdk==1.31.0
  - azureml-interpret==1.31.0
  - azureml-tensorboard==1.31.0
  - azureml-mlflow==1.31.0
  - mlflow
  - sklearn-pandas
- pandas
- numpy
- tqdm
- scikit-learn
- matplotlib
- hdbscan
- umap-learn
name: azureml_489f82068b0f5a9d2c5dd8405ff38c63
```

In [34]:

```
# These dependencies can be saved to a yaml file
# from there you can test on a local conda environment to check it builds correctly
# conda env create -f /path/to/yaml.yml
clustering_env_dep.save(os.path.join(path, 'clustering_env.yml'))

# We could then use this file in the future...
# myenv = Environment.from_conda_specification(name = "myenv",
#                                              file_path = "path-to-conda-specification-file")

# Register the environment with our workspace
clustering_env.register(ws)

# confirm all is as expected
clustering_env = ws.environments['Clustering']
print(clustering_env.name)
print(clustering_env.python.conda_dependencies.serialize_to_string())
```

```
Clustering
channels:
```

```

- anaconda
- conda-forge
dependencies:
- python=3.6.2
- pip:
  - azureml-core==1.31.0
  - azureml-defaults==1.31.0
  - azureml-telemetry==1.31.0
  - azureml-train-restclients-hyperdrive==1.31.0
  - azureml-train-core==1.31.0
  - azureml-widgets==1.31.0
  - azureml-pipeline-core==1.31.0
  - azureml-pipeline-steps==1.31.0
  - azureml-opendatasets==1.31.0
  - azureml-automl-core==1.31.0
  - azureml-automl-runtime==1.31.0
  - azureml-train-automl-client==1.31.0
  - azureml-train-automl-runtime==1.31.0
  - azureml-train-automl==1.31.0
  - azureml-train==1.31.0
  - azureml-sdk==1.31.0
  - azureml-interpret==1.31.0
  - azureml-tensorboard==1.31.0
  - azureml-mlflow==1.31.0
  - mlflow
  - sklearn-pandas
- pandas
- numpy
- tqdm
- scikit-learn
- matplotlib
- hdbscan
- umap-learn
name: azureml_b681d519612259e621bd46c3500211cd

```

In [35]:

```

# Now we create our new experiment in AzureML
# create a folder for the scripts
experiment_folder_hdbscan = './embedding-wants-hdbscan-hyperdrive'
os.makedirs(experiment_folder_hdbscan, exist_ok=True)

print('The script folder has been created.')

# set up the experiment in AzureML
experiment_hdbscan = Experiment(workspace = ws,
                                 name      = 'embedding-wants-hdbscan-hyperdrive')

print("Experiment name : " + experiment_hdbscan.name)

```

The script folder has been created.
Experiment name : embedding-wants-hdbscan-hyperdrive

Create HDBSCAN experiment

Having specified an environment including the relevant package, we can now script the experiment itself

In [36]:

```

%%writefile $experiment_folder_hdbscan/hdbscan_exploration.py
# Above line writes this code chunk as a script, it will be used by the compute for each run of the experiment
# It gets written to the 'experiment_folder' in the cloud
print("importing packages")
# import basics
# all included in base python, no need for conda dependency
import os
import argparse
import time
import json
import statistics as st

# import AzureML
# these are conda dependencies to be included in environment specification
# Best to adapt an existing AzureML environment to ensure these are captured properly
import azureml.core
from azureml.core import Workspace, Experiment, Environment, Dataset, Run

# import tools for working with the data
# these are conda dependencies to be included in environment specification
import numpy as np
import pandas as pd
import hdbscan

# We get the Latent_space and Layer sizes for each individual experiment from the file args of this python file
parser = argparse.ArgumentParser()

# the datafolder to access the dataset is the most fiddly argument, involving some magic and numerous problems
# We pass to this script a 'dataset.as_mount()' object, via ScriptRunConfig, but this script receives a string
# That string would be presented to this script as a string variable,

```

```

# eg $AZUREML_DATAREFERENCE_GUIDHERE, which refers to a temp folder on the compute instance
# https://azure.github.io/azureml-examples/docs/cheatsheet/data/
print("Getting args")
parser.add_argument('--datafolder',      type=str, dest='data_folder')
parser.add_argument('--minclustersize',   type=int, dest='min_cluster_size')
parser.add_argument('--minsamples',       type=int, dest='min_samples')

# get the args
args = parser.parse_args()

# convert args to variables in this script
data_folder      = args.data_folder
min_cluster_size = args.min_cluster_size
min_samples      = args.min_samples

# Get the experiment's currently available 'run' context
# A run represents a single trial of an experiment.
# Runs are used to monitor the asynchronous execution of a trial, log metrics and store outputs
run = Run.get_context()
ws  = run.experiment.workspace # Workspace.from_config()
print("Workspace:", ws.name)

# get the data
train_path = os.path.join(data_folder, 'embeddings_train_un_enc_umap_30.npy')
embeddings_train_enc_umap = np.load(file=train_path)
print("Got data")
##### HDBSCAN #####
# create clustering object
clusterer = hdbscan.HDBSCAN(metric          = 'manhattan',
                             min_cluster_size= int(min_cluster_size),
                             min_samples     = int(min_samples))

# cluster the data!
print("Starting clustering analysis")
start_time = time.time()
clusterer.fit(embeddings_train_enc_umap)
end_time = time.time()

# get results
cluster_labels = clusterer.labels_
cluster, count = np.unique(cluster_labels, return_counts=True)

# remove details of non-clustered points
print("Wrangling clusters")
if cluster[0] == -1:
    if len(cluster) > 1:
        cluster= cluster[1:]
        count  = count[1:]
    else:
        cluster= []
        count  = [np.nan]

# calculate metrics
print("Calculating metrics")
cluster_qty      = len(cluster)
count_mean       = st.mean(count)
propn_clustered = sum(count)/embeddings_train_enc_umap.shape[0]
duration_mins   = (end_time - start_time)/60

# Log the metrics
print("Logging metrics")
run.log("cluster_qty",    int(cluster_qty))
run.log("count_mean",     float(count_mean))
run.log("propn_clustered",float(propn_clustered))
run.log("duration_mins",  float(duration_mins))

# files saved in the "./outputs" folder are automatically uploaded into run history
print("Saving model")
os.makedirs('./outputs/clusters', exist_ok=True)
run_name = "Clusters_minclustersize_" + str(min_cluster_size) + "_minsamples_" + str(min_samples)

# save the clusters
filename = os.path.join('./outputs/clusters', run_name+'.npy')
np.save(file=filename, arr=cluster_labels)

# end run
run.complete()

```

Overwriting ./embedding-wants-hdbscan-hyperdrive/hdbscan_exploration.py

In [37]:

```

# set up the cluster, don't need gpu this time

# choose a name for your cluster
cluster_name = 'Hyperdrive-03'

# If compute already exists, use it, else create

```

```

try:
    cluster_hdbscan = ComputeTarget(workspace=ws, name=cluster_name)
    print('Found existing compute target')

except ComputeTargetException:
    print('Creating a new compute target...')
    # this experiment needs lots of memory, 28GB has been provisioned for each instance in the Standard_DS12_v2
    # unfortunately this has 4 cores, which we don't need.
    # The number of concurrent cores we have in this subscription is just 12, so only three VM's can run concurrently.
    compute_hdbscan_config = AmlCompute.provisioning_configuration(vm_size = 'Standard_DS12_v2',
                                                                    max_nodes= 6)
    # create the cluster
    cluster_hdbscan = ComputeTarget.create(ws, cluster_name, compute_hdbscan_config)

# can poll for a minimum number of nodes and for a specific timeout.
# if no min node count is provided it uses the scale settings for the cluster
cluster_hdbscan.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)

# use get_status() to get a detailed status for the current cluster.
print(cluster_hdbscan.get_status().serialize())

```

```

Found existing compute target
Succeeded
AmlCompute wait for completion finished

```

```

Minimum number of nodes requested have been provisioned
{'currentNodeCount': 0, 'targetNodeCount': 0, 'nodeStateCounts': {'preparingNodeCount': 0, 'runningNodeCount': 0, 'idleNodeCount': 0, 'unusableNodeCount': 0, 'leavingNodeCount': 0, 'preemptedNodeCount': 0}, 'allocationState': 'Steady', 'allocationStateTransitionTime': '2021-07-06T23:32:19.821000+00:00', 'errors': None, 'creationTime': '2021-04-30T15:16:47.666029+00:00', 'modifiedTime': '2021-04-30T15:17:03.020533+00:00', 'provisioningState': 'Succeeded', 'provisioningStateTransitionTime': None, 'scaleSettings': {'minNodeCount': 0, 'maxNodeCount': 6, 'nodeIdleTimeBeforeScaleDown': 'PT120S'}, 'vmPriority': 'Dedicated', 'vmSize': 'STANDARD_DS12_V2'}

```

In [38]:

```

# Set up the hyperdrive hyperparam search

# Rule of thumb, we should expect sqrt(N) clusters in a space with N points.
# N=30k in our sample, sqrt(N) = 173. Let's say 200.
# Hence, we also get sqrt(N) points per cluster
# [ because clusters * points_per_cluster = N. So, sqrt(n) * sqrt(N) = N ]
# So we'll hunt around sqrt(N) = points_per_cluster, and lower.
# We expect very sparse data, so lower our standards for clustering.
# HDBSCAN also let's us set the min_samples, which is typically same as points_per_cluster
# But the algorithm is more sensitive when min_samples is less than points_per_cluster
params_hdbscan = GridParameterSampling({ '--minclustersize': choice(1000, 2000, 3000, 4000),
                                         '--minsamples' : choice( 250, 500, 1000, 2000)})

config_hdbscan = ScriptRunConfig(source_directory= experiment_folder_hdbscan,
                                  script      = 'hdbscan_exploration.py',
                                  arguments   = ['--datafolder', embeddings_enc_umap_dataset.as_mount()], #'tmp\data'
                                  compute_target = cluster_hdbscan,
                                  environment = clustering_env)

hyperdrive_hdbscan = HyperDriveConfig(run_config          = config_hdbscan,
                                       hyperparameter_sampling= params_hdbscan,
                                       policy                = None, # early_termination_policy
                                       primary_metric_name   = 'proprn_clustered',
                                       primary_metric_goal   = PrimaryMetricGoal.MAXIMIZE,
                                       max_total_runs        = 16,
                                       max_concurrent_runs   = 3)

# Execute the experiment with the hyperdrive config
hyperdrive_hdbscan_run = experiment_hdbscan.submit(config=hyperdrive_hdbscan)

# Show the status
RunDetails(hyperdrive_hdbscan_run).show()
hyperdrive_hdbscan_run.wait_for_completion()

```

```

Out[38]: {'runId': 'HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd',
          'target': 'Hyperdrive-03',
          'status': 'Completed',
          'startTimeUtc': '2021-07-07T16:08:42.458398Z',
          'endTimeUtc': '2021-07-07T16:51:52.861174Z',
          'error': {'error': {'code': 'UserError',
                             'message': 'User errors were found in at least one of the child runs.'},
                    'messageParameters': {},
                    'details': []},
          'time': '0001-01-01T00:00:00.000Z'},
          'properties': {'primary_metric_config': '{"name": "proprn_clustered", "goal": "maximize"}',
                         'resume_from': 'null',
                         'runTemplate': 'HyperDrive',
                         'azureml.runsource': 'hyperdrive',
                         'platform': 'AML',
                         'ContentSnapshotId': '012b28a2-9727-41e0-ae67-e3762dab2ee0',
                         'score': '0.8810883140053524',
                         'best_child_run_id': 'HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_11',
                         'best_metric_status': 'Succeeded'},

```

```
'inputDatasets': [],
'outputDatasets': [],
'logFiles': {'azureml-logs/hyperdrive.txt': 'https://autoencwstoraged8ea79dd7.blob.core.windows.net/azureml/ExperimentRun/dci.d.HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd/azureml-logs/hyperdrive.txt?sv=2019-02-02&sr=b&sig=gv3y16LY%2B0nEgwUb7HboFXi5cxm8B0sqhos5a6Y5Woc%3D&st=2021-07-07T16%3A42%3A09Z&se=2021-07-08T00%3A52%3A09Z&sp=r'},
'submittedBy': 'Morris Corfield'}
```

We want the hyper params which cluster the majority of points and provide a useful number of clusters, not just 2, but not hundreds. There are at least two such hyperparams in the above list, index 8 and index 12.

How do we choose between them? Are the clusters they find very similar? This is a question of categorical correlation. So we use Cramers V

In [39]:

```
# get experiment results
ws = Workspace.get(name = 'Autoenc_ws',
                    subscription_id = 'GUID',
                    resource_group = 'AzureML_PAYG')

experiment_hdbscan = Experiment(workspace = ws,
                                 name = 'embedding-wants-hdbscan-hyperdrive')

experiment_hdbscan_results = get_last_experiment_results(experiment = experiment_hdbscan,
                                                       sort_by_metric = 'propn_clustered',
                                                       minimise = False)

experiment_hdbscan_results
```

Out [39]:

	runId	minclustersize	minsamples	cluster_qty	count_mean	propn_clustered	duration_mins
4	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_11	4000	1000	3.0	108647.0	0.881088	4.025742
8	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_7	4000	500	9.0	27880.0	0.678290	3.253928
12	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_3	4000	250	9.0	26652.0	0.648436	2.736614
13	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_0	1000	250	17.0	14016.0	0.644103	2.739685
14	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_2	3000	250	11.0	21449.0	0.637810	2.737381
9	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_6	3000	500	12.0	17697.0	0.574073	3.211960
15	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_1	2000	250	16.0	12955.0	0.560349	2.808186
6	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_9	2000	1000	13.0	15484.0	0.544160	4.964052
10	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_5	2000	500	17.0	11768.0	0.540827	3.262002
5	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_10	3000	1000	11.0	18152.0	0.539756	4.018949
11	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_4	1000	500	23.0	8581.0	0.533569	3.181019
7	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_8	1000	1000	19.0	10368.0	0.532528	4.022460
0	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_15	4000	2000	NaN	NaN	NaN	NaN
1	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_14	3000	2000	NaN	NaN	NaN	NaN
2	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_13	2000	2000	NaN	NaN	NaN	NaN
3	HD_71dc30bd-ff1f-4315-bfeb-8d6cc7f40abd_12	1000	2000	NaN	NaN	NaN	NaN

Clustering Results

The majority of params give us clusters with assimilate between half and nearly 90% of all the data. The most successful clustering takes in 88% and gives 3 clusters. Russell himself divides his data into two groups, insatiable and satiable desires. However, it will be difficult to make useful generalisations about 108,000 sentences in each cluster, which is where that most successful model lies. It will be more informative to analyse many smaller clusters. We can review how, or even whether, these generalise into larger clusters at a later point.

Therefore, we'll investigate the clustering which includes most of the data, 68%, but offers a useful number of clusters, 9, for us to consider what each cluster represents.

In [40]:

```
# identify the required clustering
minclustersize_a= 4000
minsamples_a = 500
query_text = "(minclustersize=='"+str(minclustersize_a)+"') and (minsamples=='"+str(minsamples_a)+"')"
chosen_run_id = experiment_hdbscan_results.query(query_text)
chosen_run_id = chosen_run_id['runId'].iloc[0]
chosen_run = Run(experiment=experiment_hdbscan, run_id=chosen_run_id)
output_filename = 'Clusters_minclustersize'+str(minclustersize_a)+'_minsamples'+str(minsamples_a)+'.npy'

# download the associated clustering assignments
def download_clusters(chosen_run, output_filename):
    output_file_path=''
    for filename in chosen_run.get_file_names():
        if 'Clusters' in filename:
            output_file_path = os.path.join(path, output_filename)
```

```

else:
    pass

if output_file_path != '':
    print('Downloading from {} to {} ...'.format(filename, output_file_path))
    chosen_run.download_file(name=filename, output_file_path=output_file_path)

output_file_path = ''

# use the download function...
download_clusters(chosen_run      = chosen_run,
                  output_filename = output_filename)

# get the clusters into memory
cluster_hdbscan = np.load(os.path.join(path, output_filename))

```

Downloading from outputs/clusters/Clusters_minclustersize_4000_minsamples_500.npy to /home/oliver/Documents/Deep/Data/Clusters_minclustersize4000_minsamples500.npy ...

Are The Clusters Just Searchword Clusters?

We have some encouraging clusters, but is HDBSCAN simply forming clusters around the searchwords? For example, a cluster for 'want', another cluster for 'wise', etc.

We can quickly check whether the cluster assignments correlate with the searchword found in the text

In [41]:

```

# We need to know which searchword was found in each text...

# get data
sentences_train = pd.read_csv(os.path.join(path,'sentences_train.csv'), sep='~', encoding='ascii' )

# get searchwords
searchwords = [' wanted', ' wants', ' want', ' desired', ' desires', ' desire', ' wished', ' wishes', ' wish']

# find searchwords in texts
matches = pd.DataFrame()
for word in searchwords:
    matches[word] = sentences_train['text'].str.find(word)
matches = matches.to_numpy()

# get index of first word which is matched
matches_first = np.argmax(matches > 0, axis=1)

# add results to data
sentences_train['FirstMatchIndex'] = matches_first
sentences_train['FirstMatch']      = [searchwords[index] for index in matches_first]

# save results
sentences_train.to_csv(path_or_buf=os.path.join(path,'sentences_train.csv'), index=False, sep='~', encoding='ascii')

# display sample to screen for common sense check
display(sentences_train.sample(n=6))

```

	wikidata_id	sentence_ref	clause_ref	text	text_idxs	searchword_tkn_idx	searchwords	mask	FirstMatchIndex	FirstMatch
57610	Q13365595	204	0	Political parties are not necessary; all other...	[2, 675, 2575, 50, 52, 2378, 73, 65, 89, 2357,...]	[18]	['wishes']	True	7	wishes
224277	Q4381750	34	-1	Thus keeping up with protests is a way they ca...	[2, 1086, 2882, 71, 29, 7826, 25, 21, 161, 59,...]	[18]	['wants']	True	1	wants
372020	Q494312	36	1	the customers select the pictures that they wi...	[2, 14, 5279, 5407, 14, 3104, 30, 59, 2536, 20...]	[8]	['wish']	True	8	wish
417921	Q4900911	10	1	rather she desired her temple to be erected at...	[2, 864, 39, 9264, 36, 1478, 20, 44, 6184, 35,...]	[3]	['desired']	True	3	desired
177050	Q60747975	18	1	some players wanted a less tame "Hell's Kitche...	[2, 109, 1007, 417, 21, 787, 21950, 13, 7, 112...]	[3]	['wanted']	True	0	wanted
42807	Q949746	22	-1	He wanted to attract tourists as well as to pr...	[2, 24, 417, 20, 7637, 9215, 28, 134, 28, 20, ...]	[2]	['wanted']	True	0	wanted

In [42]:

```
# Get Cluster Assignments and FirstMatchIndexes together on one dataset
sentences_train_masked = sentences_train.query("mask==True").copy()
sentences_train_masked['cluster_hdbscan'] = cluster_hdbscan

# Then run Cramers V is see if there is any correlation between these two factors
# Reference : https://towardsdatascience.com/the-search-for-categorical-correlation-a1cf7f1888c9

def cramers_v(x, y):
    confusion_matrix = pd.crosstab(x,y)
    chi2 = ss.chi2_contingency(confusion_matrix)[0]
    n = confusion_matrix.sum().sum()
    phi2 = chi2/n
    r,k = confusion_matrix.shape
    phi2corr = max(0, phi2-((k-1)*(r-1))/(n-1))
    rcorr = r-((r-1)**2)/(n-1)
    kcorr = k-((k-1)**2)/(n-1)
    return np.sqrt(phi2corr/min((kcorr-1),(rcorr-1)))

print("Correlation between clusters and searchwords. 1=Perfectly Correlated. 0=No Relationship")
print(cramers_v(sentences_train_masked['cluster_hdbscan'],
               sentences_train_masked['FirstMatchIndex']))
```

Correlation between clusters and searchwords. 1=Perfectly Correlated. 0=No Relationship
0.16702216519243204

The correlation between searchwords and the clusters is low. This is encouraging, the model has discovered more than just the searchwords we entered in the first place.

Do Other HDBSCAN Parameters Give Very Different Clusters?

We'd like to know whether the choice of min_cluster_size and min_samples gives us approximately the same clusters or very different clusters. We know that it gives different number of clusters, so there will be differences. However, are the 10 clusters for min_cluster_size=4000 and min_samples=500 completely different to the 12 clusters identified for min_cluster_size=3000 and min_samples=1000?

In other words, do the two clustering closely correlate? Again, we can use Cramers V for this task...

In [43]:

```
# identify the required clustering
minclustersize_b= 4000
minsamples_b      = 250
query_text        = "(minclustersize=='"+str(minclustersize_b)+"') and (minsamples=='"+str(minsamples_b)+"')"
chosen_run_id     = experiment_hdbscan_results.query(querytext)
chosen_run_id     = chosen_run_id['runId'].iloc[0]
chosen_run        = Run(experiment=experiment_hdbscan, run_id=chosen_run_id)
output_filename   = 'Clusters_minclustersize'+str(minclustersize_b)+'_minsamples'+str(minsamples_b)+'.npy'

# download the associated clustering assignments
download_clusters(chosen_run      = chosen_run,
                  output_filename = output_filename)

# get the clusters into memory
cluster_hdbscan_compare = np.load(os.path.join(path, output_filename))

# CramersV
print("Correlation between clusters for minclustersize=",str(minclustersize_a)," , min_samples=",minsamples_a," and")
print("                                minclustersize=",str(minclustersize_b)," , min_samples=",minsamples_b)
print("1=Perfectly Correlated. 0=No Relationship")
print(cramers_v(cluster_hdbscan, cluster_hdbscan_compare))
```

Downloading from outputs/clusters/Clusters_minclustersize_4000_minsamples_250.npy to /home/oliver/Documents/Deep/Data/Clusters_minclustersize4000_minsamples250.npy ...
Correlation between clusters for minclustersize= 4000 , min_samples= 500 and
minclustersize= 4000 , min_samples= 250
1=Perfectly Correlated. 0=No Relationship
0.8838969462700199

There is high correlation between clusters generated using different parameters for HDBSCAN. This is encouraging as it implies the model has identified something about the data irrespective of the chosen clustering parameters.

Characterising Each Cluster (Topic Modelling)

We need to characterise these clusters. This is a case of Topic Modelling where we use the model's latent space to find words which characterise each cluster. But, before we do that, let's inspect samples from each cluster.

We will attempt to find the mid-point (aka centroid) of each cluster and then sample points around that centroid.

In [44]:

```
# we will attempt to find centroids using embeddings in 60dim space,
# although they were calculated from embeddings in 30dim space (derived by UMAP from 60dim space)
embeddings_train_enc = np.load(os.path.join(path, 'embeddings_train_enc.npy'))
```

```
# stack the cluster assignment next to the embeddings
# this gives us one numpy of all embeddings and their assigned cluster
embeddings_train_enc_clustered = np.hstack((cluster_hdbscan[:, np.newaxis],
                                             embeddings_train_enc))
```

In [45]:

```
# find centroids of each cluster
cluster_pd = pd.DataFrame({'cluster': np.unique(cluster_hdbscan)})
median_list = []

for cluster in cluster_pd.iterrows():
    # for euclidean we would find the mean of each cluster
    # but for manhattan distances (ie many dimensions), we seek median

    cluster_mask = embeddings_train_enc_clustered[:, 0] == cluster[1].cluster
    cluster_embeds = embeddings_train_enc_clustered[cluster_mask, 1:]
    cluster_median = np.median(cluster_embeds, axis=0)

    # record the median
    median_list.append(cluster_median)

cluster_pd['cluster_median'] = median_list

# view sample
cluster_pd.head(n=3)
```

Out[45]:

	cluster	cluster_median
0	-1	[0.09721057116985321, -0.18499122560024261, 0....
1	0	[-0.011794622987508774, 0.3272390067577362, 0....
2	1	[0.12614373862743378, 0.6808471083641052, -0.0...

Now that we have centroids, we can seek nearby points and then display the sentences associated with those points. Each cluster is at least 3,000 points. So, to get a feel for each cluster we will identify the 100 points closest to the centroid and randomly sample 5 of those.

In [46]:

```
# bring useful data to one dataframe
embeddings_dists = pd.DataFrame({'text' : sentences_train_masked['text'],
                                 'cluster' : embeddings_train_enc_clustered[:, 0],
                                 'embedding': [np.array(row) for row in embeddings_train_enc_clustered[:, 1:]]})

# the below row gets the median for each embedding's cluster
embeddings_dists = cluster_pd.join(embeddings_dists.set_index('cluster'), on='cluster')

# get manhattan distances between the embeddings and their cluster's median (aka centroid)
distance_to_median = [distance.cityblock(a, b) for a, b in zip(embeddings_dists['cluster_median'],
                                                               embeddings_dists['embedding'])]
embeddings_dists['distance_to_median'] = distance_to_median

# rank distances to cluster medians
embeddings_dists['rank_distance_to_median'] = embeddings_dists.groupby('cluster')[['distance_to_median']].rank('first')

# lets see our data..
display(embeddings_dists.head(n=3))
```

	cluster	cluster_median	text	embedding	distance_to_median	rank_distance_to_median
0	-1	[0.09721057116985321, -0.18499122560024261, 0....	it may be assumed that the cardinals wanted to...	[0.10834327340126038, -0.8357318639755249, 0.6...	24.550987	773.0
0	-1	[0.09721057116985321, -0.18499122560024261, 0....	the Danbury Ice Arena announced that it did no...	[0.6275748014450073, 0.7949279546737671, 0.618...	36.629433	68230.0
0	-1	[0.09721057116985321, -0.18499122560024261, 0....	the king announced that he wanted to go to Porto	[0.43178173899650574, -0.48249733448028564, 0....	30.550915	19009.0

In [49]:

```
## Proceed to select samples

# get a random sample of points for which we will seek nearest neighbours
n_neighbours = 4
rank_limit = 30

# Sample points closest to each cluster's median
cluster_examples = []
for cluster in cluster_pd.iterrows():
    # get top items (by dist to median) within cluster
    query_text = "(rank_distance_to_median<=" + str(rank_limit) + ") and (cluster==" + str(cluster[1].cluster) + ")"
    shortlist = embeddings_dists.query(query_text)
    # sample from within those top items
    shortlist_sample = shortlist.sample(n=n_neighbours)
    # save data to list
    cluster_examples.append(shortlist_sample)
```

```

# concatenate all samples into one dataframe
cluster_examples = pd.concat(cluster_examples)

# present the samples from each cluster
pd.set_option('display.width', 2000)
pd.set_option('display.max_colwidth', 2000)

for cluster in cluster_pd.itertuples():
    if cluster.cluster > -1: # ignore the unclustered clauses, which are labelled '-1'.
        print("CLUSTER = ", str(cluster.cluster))
        texts = cluster_examples.query("cluster=="+str(cluster.cluster))[[['text']]].reset_index(drop=True)
        display(texts)

```

CLUSTER = 0

text

- 0** Al-Raed wanted to renew the loan because of his great form
- 1** Lingo originally wanted the team to play out of LaRue
- 2** Dunkerque wanted to sign him but would not meet RFCL's fee
- 3** Lingo wanted the team to help run his kennels in addition to playing football

CLUSTER = 1

text

- 0** Mike now says that he wants Mary Ann to stay there
- 1** She wants him to forget about the doll and just leave
- 2** but what she really wants is to assume their house
- 3** but Lara does not want to accept anything from him

CLUSTER = 2

text

- 0** who wants the grandfather's heritage to be kept secret
- 1** who wants him to join the Starling Guild
- 2** who wanted him to be a watchmaker
- 3** who wanted her to give it another shot

CLUSTER = 3

text

- 0** Novik had always wanted an FM station to accompany WPUT
- 1** Greenblatt wanted a show that had a style similar to the Indiana Jones movies
- 2** Mechanic wanted the trio to be involved in the new film
- 3** The band then learnt that Steve Albini wanted to work with Kill Kenada

CLUSTER = 4

text

- 0** Wrongly guessing that she wants a larger place to live
- 1** since he did not want them to go to Palestine
- 2** as he wanted to draw attention to the fact Carter was breaking the law
- 3** In defiance of Diggs' wish to keep the girls

CLUSTER = 5

text

- 0** but you do want to see something like a kiss"
- 1** I guess it doesn't want me"
- 2** I don't want to talk about her"
- 3** I didn't want to ask for more"

CLUSTER = 6

text

- 0** In November 2000, Lees appeared on Stars in Their Eyes where he sang Aerosmith's "I Dont Want to Miss a Thing"
- 1** In 2011, he featured in a music video for the song 'I Want You to Remember' by solo artist I Am Spartacus
- 2** Frey said that he and Henley wanted the song "to open like an episode of the Twilight Zone"

3 According to Voris: NEWLINE It was actually an old Sayles script that some producers bought and wanted to make for action star Dolph Lundgren

CLUSTER = 7

text

- 0 the President of The Shire of Carnarvon stated that he did not want to place blame on any agency
- 1 He wanted to limit the marketing of this product to certain geographic regions
- 2 Krist argues Duke Stephen wanted to ensure to Transylvania a high degree of autonomy
- 3 they simply wanted to rid the countryside of Bolshevik influence by any means necessary

CLUSTER = 8

text

- 0 Sock said: "That was in if you want to challenge it
- 1 further elaborating that "we want to do more with the band
- 2 and set the stage for his decisions later in the series: Norris opined that "Hank wants a clean soul
- 3 Rahiem has said: "We just want to make wholesome

Following inspection the above clusters could be summarised as:

- 0 Desires in sports
- 1 Desire to be distanced from someone
- 2 Desires of others (who wanted...)
- 3 Desires in acting
- 4 Desires about other people's lives
- 5 Obstinacy on part of third party
- 6 Desires in music
- 7 Desires relating to place or territory
- 8 [Unclear]

Instinctively these feel like plausible desires although odd candidates for the most fundamental desires in people. It is clear that these are all 'insatiable' desires, as Russell would have classed them. There is no cluster for satiable desires such as wanting to eat, sleep, drink etc. The dataset we chose, Wikipedia, may be partly to blame for this. Of course, this summary is nice but ultimately it is unsatisfactory because:

- a. it is not objective
- b. it is based only on small samples of very large clusters

We would like an objective way to identify words to represent these clusters. There is an approach which selects the most common words in each cluster which are not commonly found in other clusters. See below link for full description of 'TF-IDF':

<https://towardsdatascience.com/topic-modeling-with-bert-779f7db187e6>

In [50]:

```
# let's save our data to date
embeddings_dists.to_csv(path_or_buf=os.path.join(path, 'embeddings_dists.csv'), index=False, sep='~', encoding='ascii')
```

In [51]:

```
# unite the texts of each cluster into one document per cluster.
docs_df      = embeddings_dists.query("cluster != '-1'")[['text', 'cluster']]
docs_df['text_id'] = range(len(docs_df))
docs_per_topic = docs_df.groupby(['cluster'], as_index = False).agg({'text': ' '.join})

# view one example
docs_per_topic.iloc[2,:]
```

Out[51]: cluster

```
1
text      I Wish I Knew How It Would Feel to Be) Free/One Bernie Ecclestone did not want to have 90 The Revolution will not b
e delayed: follow me, if you want my next desire was to write a work Phil and Claire celebrate their 20th wedding anniversary
and they both want to surprise each other with their gifts Phil feels frustrated to learn that Claire has loved Harold's singi
ng and does not want to sing anymore Alex agrees to help him but Sienna is also the girl who Alex wants to make her best frien
d her moral convictions and desire to be compassionate are seriously questioned when she finds out that the teenage driver kil
led a classmate He finds a man who was not looking for adventures but rather wanted to act as a mediator between East and West
We learned that if you really wanted a movie to come out the way you wanted it to Her mum was disowned by her parents after sh
e chooses to marry an American against their wishes she falls ill and wishes to reunite with her family but she chose to leave
the show because she wanted to take some weeks off Soon after arriving in Japan they are helped by a rival veteran crime boss
who wants to oust Killer the gangster for good and dominate his operations she was always sure that she wanted to pursue theat
re as a profession while Ged and Rob wanted to use Venoms management Wallington grew up in Pennsylvania and always wanted to b
e a writer but was not sure how to go about it but now wanted to settle down and get his life in order I don't want to mess firs
t and foremost that you want to hide it I just knew that I wanted to act and wished for a bible she could read herself I want
to be particular about that it's just a slice of it He knew then he wanted to make music his career I want to get them to belie
```

ve they can do it even when times are getting tough Ryan's demands are all the more terrifying because he doesn't w...
Name: 2, dtype: object

In [52]:

```
from sklearn.feature_extraction.text import CountVectorizer

# importance value, based on frequency, for each word in a cluster
def c_tf_idf(documents, m, ngram_range=(1, 1)):
    count = CountVectorizer(ngram_range=ngram_range, stop_words="english").fit(documents)
    t = count.transform(documents).toarray()
    w = t.sum(axis=1)
    tf = np.divide(t.T, w)
    sum_t = t.sum(axis=0)
    idf = np.log(np.divide(m, sum_t)).reshape(-1, 1)
    tf_idf = np.multiply(tf, idf)

    return tf_idf, count

tf_idf, count = c_tf_idf(docs_per_topic.text.values, m=len(embeddings_dists))
```

In [53]:

```
# score words within each cluster by their frequency
def extract_top_n_words_per_topic(tf_idf, count, docs_per_topic, n=20):
    words = count.get_feature_names()
    labels = list(docs_per_topic.cluster)
    tf_idf_T = tf_idf.T
    indices = tf_idf_T.argsort()[:, -n:]
    top_n_words = {label: [(words[j], tf_idf_T[i][j]) for j in indices[i]][::-1] for i, label in enumerate(labels)}
    return top_n_words

top_n_words = extract_top_n_words_per_topic(tf_idf, count, docs_per_topic, n=20)

# convert top_n_words to pandas for convenience
top_n_words_pd = []
for cluster in top_n_words:
    word_freq = pd.DataFrame(top_n_words[cluster])
    word_freq['cluster'] = [cluster]*len(word_freq)
    word_freq.columns = ['word', 'freq', 'cluster']
    top_n_words_pd.append(word_freq)
top_n_words_pd = pd.concat(top_n_words_pd)

# create mask to remove the top_n_words which are simply the searchwords we originally specified
searchwords = ['wanted', 'wants', 'want', 'desired', 'desires', 'desire', 'wished', 'wishes', 'wish']
mask = [word not in searchwords for word in top_n_words_pd['word']]

# print the top 12 words per cluster
for cluster in top_n_words:
    top_n_words_cluster = top_n_words_pd.iloc[mask, :].query("cluster=="+str(cluster)).reset_index(drop=True)
    top_n_words_cluster = top_n_words_cluster['word'].tolist()
    top_n_words_cluster = top_n_words_cluster[0:12]
    if cluster > -1:
        print("Cluster ", cluster, ":", *top_n_words_cluster)
```

```
Cluster 0 : team club play football league season manager coach stated expressed did players
Cluster 1 : tells does just didn't don make doesn't did father know marry life
Cluster 2 : did does know make life really doesn't marry like use man people
Cluster 3 : film band album record originally music role release play make series stated
Cluster 4 : did saying despite does telling people life stating make family father
Cluster 5 : don just saying didn't make did like people really stating know said
Cluster 6 : album released film song band single music series songs release video new
Cluster 7 : did government people new use said state students war party time
Cluster 8 : said don saying stated people make just like didn't really wrote told
```

The above results suggest a number of the clusters are simply subject orientated, the common themes of desire are swamped by the context in which it happens. For example, a deeper understanding of desires would say ambition in sport or music was simply ambition, regardless of the subject area. Whereas the above clusters clearly separate out music, sport and acting.

We need to approach clustering differently.

Delete All Resources in AzureML

The work in AzureML is now complete, all useful data and models have been saved locally. Therefore it is time to delete the workspace and all associated resources in the cloud. Any resources remaining in the cloud will continue to incur charges, not least storage charges, which are substantial for cloud SSD's used for storing training data.

In []:

```
# Delete resources in AzureML
ws.delete(delete_dependent_resources=True)
```

APPROACH 2. GloVe Embeddings

Attributes Vs Relationships

Peter Turney is one of the most influential researchers in NLP (natural language processing). In his 2006 paper "Similarity of Semantic Relations" [<https://direct.mit.edu/coli/article/32/3/379/1915/Similarity-of-Semantic-Relations>] he says:

"There are at least two kinds of similarity. Relational similarity is correspondence between relations, in contrast with attributional similarity, which is correspondence between attributes. When two words have a high degree of attributional similarity, we call them synonyms. When two pairs of words have a high degree of relational similarity, we say that their relations are analogous. For example, the word pair mason:stone is analogous to the pair carpenter:wood"

Transformers, which are the tool of choice for NLP in 2021, are well known to be excellent at finding similarity via synonyms. However, this investigation seeks relational similarity. For example, Russell's work may classify ambition as an 'insatiable desire' irrespective of that ambition relating to music or sports. We seek the similarity in the relationship regardless of the attributions. It is this project's endeavour to see how sentences about desire are clustered with respect to relationships. We discovered that transformer embeddings do indeed favour attributions and are an inappropriate tool for this task.

In the paper "Sentence Analogies: Exploring Linguistic Relationships And Regularities In Sentence Embeddings" (Zhu & deMelo 2020) [<https://arxiv.org/pdf/2003.04036.pdf>], we find a number of lessons for extracting analogies from text. Zhu & deMelo find the most successful measures of analogy using GloVe sentence embeddings, (Penning et al 2014)[<https://nlp.stanford.edu/pubs/glove.pdf>], and a measure of distance called 3CosMul, which was first presented for analogies in word embeddings in "Linguistic Regularities in Sparse and Explicit Word Representations" (Levy & Goldberg 2014)[<https://www.aclweb.org/anthology/W14-1618.pdf>]. For reference, Levy and Goldberg wanted analogies of the form: "London is to England as Baghdad is to Iraq", these are syntactic analogies not semantic analogies, nevertheless the tools can be applied to this task.

Zhu & deMelo also recommend combining the word embeddings into a sentence embedding using Discrete Cosine Transform (DCT) compression, which is a Fast Fourier Transform (FFT) tool available in the scipy package. DCT compression captures some component of the sequence of the word tokens in the sentence. They also consider simple averaging of the words tokens, and this is successful although not quite as powerful as using DCT. The precise method for applying DCT to create sentence embeddings is described in "Efficient Sentence Embedding using Discrete Cosine Transform"(Almarwani et al 2019)[<https://arxiv.org/pdf/1909.03104.pdf>], who share their code at https://github.com/N-Almarwani/DCT_Sentence_EMBEDding/blob/master/DCT.py

We can see how analogy opens up into subtle and fascinating problems in NLP, such as inference and reasoning. This is discussed in "The (Too Many) Problems of Analogical Reasoning with Word Vectors". (Rogers et al 2017)[<https://www.aclweb.org/anthology/S17-1017.pdf>], which also investigates the 3CosMul measure. Semantic analogies also inform sentiment analysis and it is a shame that transformer embeddings cannot bring analogies to that work. However, this investigation will steer clear of these wider issues for now. For reference deMelo publishes an excellent list of NLP resources for similar projects at <http://gerard.demelo.org/projects.html>

Recap

Zhu & deMelo recommend using GloVe embeddings and a function called 3CosMul to measure the distance between those embeddings. Zhu & deMelo did not attempt clustering using this combination, however this is easily done using the latest HDBSCAN package. The approach adopted in this paper is as follows:

- Convert words in each sentence to GloVe embeddings
- Identify specific embedding for the key searchword in each sentence; 'want', 'desire', 'wish' etc
 - uses Gensim package
- Apply DCT to derive sentence embedding from word embeddings
 - uses scipy fft
 - alternative is a simple average of word embeddings to form a sentence embedding
- Calculate 3CosMul score for each analogy pair, hence pairwise distance matrix
 - analogy pair = searchword A in sentence A vs searchword B in sentence B
 - eg score to what extent 'want' in 'I want food' is analogous to 'wish' in 'I wish to eat'
- Reduce dimensionality of distance matrix to 30 dims or less
 - possibly using PCA since GloVe has standard embeddings with only 50 dimensions, reduction to 30 is expected to be straightforward
- Use HDBSCAN with metric='precomputed' to cluster the pairwise distances
 - see https://hdbscan.readthedocs.io/en/latest/basic_hdbscan.html#distance-matrices
 - may need to sub sample the data as this is a combinatorial task where a 100k sentences will result in 5 billion pairwise combinations
- Analyse clusters by inspection and TF-IDF
 - If the model is truly attribute agnostic, then TF-IDF should struggle to find topics based on attributes
 - compare with clusters discovered using the transformer embedding approach

In [2]:

```
# file management
import os

# object management
import sys
```

```

# data wrangling
import glob
import pandas as pd
import numpy as np
from ast import literal_eval
from itertools import combinations, permutations

# nlp using gensim
from gensim.models import KeyedVectors
from gensim.parsing.preprocessing import preprocess_string, strip_tags, strip_punctuation
from gensim.parsing.preprocessing import strip_multiple_whitespaces, strip_numeric
from gensim.scripts.glove2word2vec import glove2word2vec

# charting and display
from matplotlib import pyplot
import plotnine as p9
import pylab
from IPython.display import Markdown, display

# dimensionality reduction
import umap

# process timing and pre-compiling
import tqdm.notebook as tq
import time
import numba

# statistical learning
import random
import math
import hdbscan
from scipy.fftpack import dct
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from scipy.spatial.distance import squareform, cityblock, cosine

# no tensorflow or pytorch required!

# set working directory
os.chdir('/home/oliver/Documents/Deep_Glove/Data')
path = os.getcwd()
print("Working path is...")
print(path)

```

```

/home/oliver/anaconda3/envs/deep_gensim/lib/python3.8/site-packages/gensim/similarities/__init__.py:15: UserWarning: The gensim.similarities.levenshtein submodule is disabled, because the optional Levenshtein package <https://pypi.org/project/python-Levenshtein/> is unavailable. Install Levenshtein (e.g. `pip install python-Levenshtein`) to suppress this warning.
    warnings.warn(msg)
Working path is...
/home/oliver/Documents/Deep_Glove/Data

```

In [9]:

```

sentences_train = pd.read_csv(os.path.join(path,'sentences_train.csv'), sep='~', encoding='ascii' )
sentences_testi = pd.read_csv(os.path.join(path,'sentences_testi.csv'), sep='~', encoding='ascii' )
sentences_valid = pd.read_csv(os.path.join(path,'sentences_valid.csv'), sep='~', encoding='ascii' )

# filter on length of sentences, mask field already exists for this usage.
sentences_train = sentences_train[ sentences_train['mask'] ].reset_index(drop=True)
sentences_testi = sentences_testi[ sentences_testi['mask'] ].reset_index(drop=True)
sentences_valid = sentences_valid[ sentences_valid['mask'] ].reset_index(drop=True)

sentences_train.name = 'train'
sentences_testi.name = 'testi'
sentences_valid.name = 'valid'

for dataset in [sentences_train, sentences_testi, sentences_valid]:
    dataset['text_idxs'] = [literal_eval(cell) for cell in dataset['text_idxs']]
    dataset['searchwords'] = [literal_eval(cell) for cell in dataset['searchwords']]
    dataset['searchwords_1st'] = [searchwords_list[0] for searchwords_list in dataset['searchwords']]

print("Training Records : ", len(sentences_train))
print("Testing Records : ", len(sentences_testi))
print("Validation Records: ", len(sentences_valid))

```

```

Training Records : 369930
Testing Records : 20740
Validation Records: 21329

```

In [7]:

```

# convert the pre-trained Glove embeddings to gensim format (word2vec)
# Glove embeddings can be downloaded from https://nlp.stanford.edu/projects/glove/

# 300 dims
filename_glove_300 = os.path.join(path, 'glove.6B.300d.txt')
filename_glove_w2v_300 = os.path.join(path, 'glove.6B.300d.word2vec.txt')

```

```
# 50 dims
filename_glove_50      = os.path.join(path, 'glove.6B.50d.txt')
filename_glove_w2v_50   = os.path.join(path, 'glove.6B.50d.word2vec.txt')
```

In [7]:

```
# convert from txt to word2vec
glove2word2vec(filename_glove_300, filename_glove_w2v_300)

glove2word2vec(filename_glove_50, filename_glove_w2v_50)
```

```
<ipython-input-7-788653a0ff5f>:2: DeprecationWarning: Call to deprecated `glove2word2vec` (KeyedVectors.load_word2vec_format(.., binary=False, no_header=True) loads GLoVE text vectors.).
<ipython-input-7-788653a0ff5f>:4: DeprecationWarning: Call to deprecated `glove2word2vec` (KeyedVectors.load_word2vec_format(.., binary=False, no_header=True) loads GLoVE text vectors.).
```

Out[7]: (400000, 50)

The next step is to pass over all datasets creating the Glove embeddings for the sentences and searchwords. This needs to be done for both the 300D Glove embeddings and 50D Glove embeddings.

In [10]:

```
# it will be useful to know the embeddings of the searchwords
searchwrd_lst = sentences_train['searchwords_1st'].drop_duplicates().tolist()

# create the list of filters (cleaning / preprocessing) to be applied to the texts
CUSTOM_FILTERS = [lambda x: x.lower(), strip_tags, strip_punctuation, strip_multiple_whitespaces, strip_numeric]
```

In [40]:

```
# function to apply dct such that a stack of word embeddings are transformed into a sentence embedding
# derived from code by Almarwani et al 2019,
# at https://github.com/N-Almarwani/DCT_Sentence_Embbedding/blob/master/DCT.py
```

```
def get_sentence_dct(sentvec, lentrans):
    # sentence is a numpy array with shape [words, dims] of word embeddings for the sentence
    # Lentrans = Length of the transform, the number of features included in each embedding
    # If each embedding length = 50 and we take 2 features from each embedding, then output embedding length = 100
    # The more features then more detail retained, but the more dimensions we must deal with later.
    # Lentrans = 2 is recommended by Almarwani, but we wish to avoid further compression later,
    # so 1 will have to suffice. Therefore, output length = word embed length
    if (sentvec.shape[0] < lentrans):
        sentvec = np.reshape(dct(sentvec, n=lentrans, norm='ortho', axis=0)[:lentrans,:], (lentrans*sentvec.shape[1],))
    else:
        sentvec = np.reshape(dct(sentvec, norm='ortho', axis=0)[:lentrans,:], (lentrans*sentvec.shape[1],))

    return sentvec
```

In [128...]

```
# Loop thru all datasets and get Glove embeddings

for dims_pair in [(50, filename_glove_w2v_50), (300, filename_glove_w2v_300)]:
    dims, model_w2v = dims_pair

    # get gensim Glove model into memory
    model = KeyedVectors.load_word2vec_format(model_w2v, binary=False)

    for dataset in [sentences_valid, sentences_train, sentences_testi]:
        # apply preprocessing

        # preprocess the data
        dataset['text_glove'] = [preprocess_string(text, CUSTOM_FILTERS) for text in dataset['text']]
        # remove words which are not present in the glove vocabulary, cannot convert to embedding
        dataset['text_glove'] = [[word for word in sentence if word in model.key_to_index.keys()]
                               for sentence in dataset['text_glove']]

        # get embeddings for each word in each sentence
        sentence_list = [[model[word] for word in sentence] for sentence in dataset['text_glove']]

        # get mean embedding for entire sentence
        embeddings_sentence_avg = np.array([np.mean(wordembeds, axis=0) for wordembeds in sentence_list])
        embeddings_sentence_dct = np.array([get_sentence_dct(np.array(wordembeds), lentrans=1)
                                             for wordembeds in sentence_list])

        # get embedding for the searchwords
        # cannot apply dct to just one embedding, no compression to be done
        # so searchwords do not have two versions, one averaged and one dct. Only one raw GloVe embedding.
        embeddings_serchwrd = np.array([model[searchword] for searchword in dataset['searchwords_1st']])

        # standardise embeddings, as per Levy & Goldberg 2014,
        # using same scaler for both sentences and searchwords so on same scale
        scaler_avg = StandardScaler().fit(np.vstack((embeddings_sentence_avg, embeddings_serchwrd)))
        embeddings_sentence_avg = scaler_avg.transform(embeddings_sentence_avg)
        embeddings_serchwrd_avg = scaler_avg.transform(embeddings_serchwrd)

        # repeat for dct compressed embeddings
        scaler_dct = StandardScaler().fit(np.vstack((embeddings_sentence_dct, embeddings_serchwrd)))
```

```

embeddings_sentence_dct = scaler_dct.transform(embeddings_sentence_dct)
embeddings_serchwrd_dct = scaler_dct.transform(embeddings_serchwrd)

# save results, avg and dct
np.save(file=os.path.join(path, 'embeddings_glove_sentence_avg_'+str(dims) + '_' + dataset.name+'.npy'),
        arr=embeddings_sentence_avg)
np.save(file=os.path.join(path, 'embeddings_glove_serchwrd_avg_'+str(dims) + '_' + dataset.name+'.npy'),
        arr=embeddings_serchwrd_avg)

np.save(file=os.path.join(path, 'embeddings_glove_sentence_dct_'+str(dims) + '_' + dataset.name+'.npy'),
        arr=embeddings_sentence_dct)
np.save(file=os.path.join(path, 'embeddings_glove_serchwrd_dct_'+str(dims) + '_' + dataset.name+'.npy'),
        arr=embeddings_serchwrd_dct)

# save the serchword short List for the training data.combinations, will be same for all datasets, so save only once
# this is the list: ['want', 'wants', 'wanted', 'wish', 'wished', etc....]
if dataset.name == 'train':
    # get embeddings for the searchword shortlist
    embeddings_serchwrd_lst = np.array([model[searchword] for searchword in serchwrd_lst])

    # save for avg
    embeddings_serchwrd_lst_avg = scaler_avg.transform(embeddings_serchwrd_lst)
    embeddings_serchwrd_lst_avg_pd = pd.DataFrame({'Word' : serchwrd_lst,
                                                    'Embedding': embeddings_serchwrd_lst_avg.tolist()})
    embeddings_serchwrd_lst_avg_pd.to_csv(path_or_buf=
                                           os.path.join(path, 'embeddings_glove_serchwrd_lst_avg_'+str(dims) + '.npy'), sep='~')

    # save for dct
    embeddings_serchwrd_lst_dct = scaler_dct.transform(embeddings_serchwrd_lst)
    embeddings_serchwrd_lst_dct_pd = pd.DataFrame({'Word' : serchwrd_lst,
                                                    'Embedding': embeddings_serchwrd_lst_dct.tolist()})
    embeddings_serchwrd_lst_dct_pd.to_csv(path_or_buf=
                                           os.path.join(path, 'embeddings_glove_serchwrd_lst_dct_'+str(dims) + '.npy'), sep='~')

```

```
In [50]: # Load the embeddings created by the above loop
embeddings_sentence_train = np.load(os.path.join(path, 'embeddings_glove_sentence_dct_300_train.npy'))
embeddings_serchwrd_train = np.load(os.path.join(path, 'embeddings_glove_serchwrd_dct_300_train.npy'))

print("train embeds shape = ", embeddings_sentence_train.shape)

train embeds shape      = (369930, 300)
```

3CosMul

The chosen measure of similarity is 3CosMul, which is described in Levy & Goldberg 2014, see eqn 13, you are encouraged to read about its derivation. It is a combination of cosine similarities. It is intended to work with GloVe embeddings and to measure the distance between analogous pairs of words, not sentences. Zhu & deMelo 2020 shows that it is also applicable to semantic analogies between sentences.

It is intriguing as a distance measure because it involves three embeddings, not two. The third embedding is that of the searchword, the axis or theme along which we wish to compare the two sentence embeddings. This should be on the same scale as the two sentence embeddings, hence the scaling of the data earlier. this invites us to consider how many other problems could be investigated with such a measure.

```
In [51]: # cosine similarity, which is used by 3CosMul
def cossim(x, y):
    numerator = np.dot(x, y)
    denominator = np.linalg.norm(x) * np.linalg.norm(y)
    return numerator / denominator

# 3CosMul inspired by Levy & Goldberg 2014, eqn 13
def ThreeCosMul(sentence_a, serchwrd_a, sentence_b, serchwrd_b, epsilon=math.pow(10, -6)):
    # 3CosMul in Levy & Goldberg is a measure of similarity. Whereas we need the inverse, a distance...
    # so denominator and numerator are reversed as compared with Levy & Goldberg
    # Note, 3CosMul returns a similarity of '1' when two sentences are the same,
    # so subtract 1 to get distance of 0 between two the same.
    # Epsilon is included simply to prevent division by zero
    numerator = cossim(sentence_b, serchwrd_a)
    denominator = cossim(sentence_b, serchwrd_b) * cossim(sentence_b, sentence_a) + epsilon
    distance = (numerator / denominator) - 1.0
    # return magnitude (abs) only
    return abs(distance)

# get 3CosMul measure given only indices of embeddings
def get_ThreeCosMulDistance(index_pair, embeddings_sentence, embeddings_serchwrd):
    index_a, index_b = index_pair
    # get embedding pair a
    sentence_a = embeddings_sentence[index_a]
    serchwrd_a = embeddings_serchwrd[index_a]
    # get embedding pair b
    sentence_b = embeddings_sentence[index_b]
    serchwrd_b = embeddings_serchwrd[index_b]
    # return distance of a vs b, magniture (abs) only.
```

```

distance = ThreeCosMul(sentence_a, serchwrd_a, sentence_b, serchwrd_b)
return abs(distance)

```

Testing the Analogy Measure

We have a measure for analogy, also we still have the traditional method of comparing two vectors in high dimensional space, the cosine distance. Before we apply 3CosMul to millions of records, let's try to build some intuition for it, as compared with cosine distance, by inspecting some basic examples. These examples will not be scaled or DCT processed. For simplicity they are raw GloVe embeddings.

```

In [13]:
# Sample data
# get gensim Glove model into memory
model = KeyedVectors.load_word2vec_format(filename_glove_w2v_300, binary=False)

In [14]:
# the first two sentences use the searchword in a similar way, the third does not.
# we expect low distance between the rows 0 and 1, high distance between rows 1 and 2.
text_sandpit = ["He wanted to become an astronaut", "They wished for a future in space",
                "His success left nothing to desire", "Their achievement left them contented",
                "You wanted him to love me", "You desired to bring us together"]
searchword_sandpit = ['wanted', 'wished', 'desire', 'contented', 'wanted', 'wished']
sandpit = pd.DataFrame({'text':text_sandpit, 'searchwords_1st':searchword_sandpit })

# data processing and embedding
# for sentences...
sandpit['text_glove'] = [preprocess_string(text, CUSTOM_FILTERS) for text in sandpit['text']]
sandpit['text_glove'] = [[word for word in sentence if word in model.key_to_index.keys()]
                        for sentence in sandpit['text_glove']]

embeddings_sentence_sandpit = np.array([ np.mean([model[word] for word in sentence], axis=0)
                                         for sentence in sandpit['text_glove'] ] )

# for searchwords
embeddings_serchwrd_sandpit = np.array( [model[searchword] for searchword in sandpit['searchwords_1st']] )

# standardise embeddings, using same scaler for both sentences and searchwords
scaler = StandardScaler().fit(np.vstack((embeddings_sentence_sandpit, embeddings_serchwrd_sandpit)))
embeddings_sentence_sandpit = scaler.transform(embeddings_sentence_sandpit)
embeddings_serchwrd_sandpit = scaler.transform(embeddings_serchwrd_sandpit)

In [15]:
# Get distance matrices, starts with getting combinations of sentences
indices_cmb = list(combinations(sandpit.index, 2))
sentences = sandpit['text'].tolist()

## 3CosMul distance matrix
distance_3cosmul_lst = [get_ThreeCosMulDistance(index_pair      = pair,
                                                embeddings_sentence = embeddings_sentence_sandpit,
                                                embeddings_serchwrd = embeddings_serchwrd_sandpit)
                        for pair in indices_cmb]

distance_3cosmul_mat = pd.DataFrame(data=squareform(distance_3cosmul_lst), columns=sentences, index=sentences)

# Cosine distance matrix
distance_cosine_lst = [cosine(embeddings_sentence_sandpit[pair[0]],
                               embeddings_sentence_sandpit[pair[1]])
                        for pair in indices_cmb]

distance_cosine_mat = pd.DataFrame(data=squareform(distance_cosine_lst), columns=sentences, index=sentences)

# Combined distance list
sentence_pairs=[]
for index, pair in enumerate(indices_cmb):
    a,b = pair
    sentence_pair = pd.DataFrame({'sentence_a': sentences[a], 'sentence_b':sentences[b]}, index=[index])
    sentence_pairs.append(sentence_pair)
sentence_pairs = pd.concat(sentence_pairs)

distance_lst = pd.DataFrame({'sentence_a' : sentence_pairs['sentence_a'],
                             'sentence_b' : sentence_pairs['sentence_b']})
distance_lst['Distance_3CosMul']= distance_3cosmul_mat
distance_lst['Rank_3CosMul'] = [int(rank) for rank in distance_lst['Distance_3CosMul'].rank()]
distance_lst['Distance_Cosine'] = distance_cosine_mat
distance_lst['Rank_Cosine'] = [int(rank) for rank in distance_lst['Distance_Cosine'].rank()]

# display results
display(Markdown('**Distance Matrix using 3CosMul distance function**'))
display(distance_3cosmul_mat)
display(Markdown('**Distance Matrix using Cosine distance function**'))
display(distance_cosine_mat)
display(Markdown('**Sentence pairs in order of 3CosMul distance**'))
display(distance_lst.sort_values(by=['Rank_3CosMul']).reset_index(drop=True))

```

	He wanted to become an astronaut	They wished for a future in space	His success left nothing to desire	Their achievement left them contented	You wanted him to love me	You desired to bring us together
He wanted to become an astronaut	0.000000	2.808327	10.075304	150.210800	21.131189	7.589306
They wished for a future in space	2.808327	0.000000	7.723172	5.896541	23.717103	3.776736
His success left nothing to desire	10.075304	7.723172	0.000000	2.680869	17.624288	4.317436
Their achievement left them contented	150.210800	5.896541	2.680869	0.000000	33.661537	7.300826
You wanted him to love me	21.131189	23.717103	17.624288	33.661537	0.000000	0.637776
You desired to bring us together	7.589306	3.776736	4.317436	7.300826	0.637776	0.000000

Distance Matrix using Cosine distance function

	He wanted to become an astronaut	They wished for a future in space	His success left nothing to desire	Their achievement left them contented	You wanted him to love me	You desired to bring us together
He wanted to become an astronaut	0.000000	0.685721	0.844014	1.009884	0.954825	0.944185
They wished for a future in space	0.685721	0.000000	0.763305	0.716545	1.104081	0.790649
His success left nothing to desire	0.844014	0.763305	0.000000	0.637920	0.898155	0.936803
Their achievement left them contented	1.009884	0.716545	0.637920	0.000000	1.076881	0.894208
You wanted him to love me	0.954825	1.104081	0.898155	1.076881	0.000000	0.707295
You desired to bring us together	0.944185	0.790649	0.936803	0.894208	0.707295	0.000000

Sentence pairs in order of 3CosMul distance

	sentence_a	sentence_b	Distance_3CosMul	Rank_3CosMul	Distance_Cosine	Rank_Cosine
0	You wanted him to love me	You desired to bring us together	0.637776	1	0.707295	3
1	His success left nothing to desire	Their achievement left them contented	2.680869	2	0.637920	1
2	He wanted to become an astronaut	They wished for a future in space	2.808327	3	0.685721	2
3	They wished for a future in space	You desired to bring us together	3.776736	4	0.790649	6
4	His success left nothing to desire	You desired to bring us together	4.317436	5	0.936803	10
5	They wished for a future in space	Their achievement left them contented	5.896541	6	0.716545	4
6	Their achievement left them contented	You desired to bring us together	7.300826	7	0.894208	8
7	He wanted to become an astronaut	You desired to bring us together	7.589306	8	0.944185	11
8	They wished for a future in space	His success left nothing to desire	7.723172	9	0.763305	5
9	He wanted to become an astronaut	His success left nothing to desire	10.075304	10	0.844014	7
10	His success left nothing to desire	You wanted him to love me	17.624288	11	0.898155	9
11	He wanted to become an astronaut	You wanted him to love me	21.131189	12	0.954825	12
12	They wished for a future in space	You wanted him to love me	23.717103	13	1.104081	15
13	Their achievement left them contented	You wanted him to love me	33.661537	14	1.076881	14
14	He wanted to become an astronaut	Their achievement left them contented	150.210800	15	1.009884	13

Distance Function Summary

Broadly the two measures, cosine and 3CosMul, agree. Notably, 3CosMul has a far larger range of distances, which may be helpful when clustering a large dataset. We will continue the analysis using 3CosMul, since it is designed specifically for the task of finding analogies. Nevertheless, it is noted that cosine distance is an approximation to 3CosMul.

Preparing the Data Shortlist

Finding analogies analytically is clearly a difficult task, so the first iteration will attempt to make the clusters easier to find by:

- a) Ensuring the searchword is the same for both sentence A and sentence B. The most common searchword is 'wanted', so the data will be filtered to those sentences where 'wanted', 'wants' or 'want' is the searchword. All three searchwords have a very similar embedding.
- b) This has the added benefit of making the 3CosMul distance calculation easier, the searchword embedding does not need to be looked up for both sentence A and sentence B. Instead we can set it to always be the embedding for 'wanted'.
- c) The analogy may be obscured in long sentences, so initial analysis will focus on sentences with less than 20 tokens (a hand picked value)
- d) The Glove embeddings in 50 dimensions are used as opposed to those in 300 dimensions. This makes the clustering easier, data in 300dims is expensive to analyse and very sparse.
- e) Using the simplest possible sentence embeddings, based on averaging the words in the sentence not DCT compression.

This still allows 145,205 examples for analysis

```
In [148...]
# Load embeddings
embeddings_sentence_train = np.load(os.path.join(path, 'embeddings_glove_sentence_avg_50_train.npy'))
embedding_serchrd_lst = pd.read_csv(os.path.join(path, 'embeddings_glove_serchrd_lst_avg_50.npy'), sep='~')
embedding_serchrd_wanted = embedding_serchrd_lst.query("Word=='wanted'")['Embedding'].iloc[0]
embedding_serchrd_wanted = np.array(literal_eval(embedding_serchrd_wanted))
print('Sentence embedding shape:', embeddings_sentence_train.shape)
print('Embedding for searchword, *wanted*, shape:', embedding_serchrd_wanted.shape)

Sentence embedding shape: (369930, 50)
Embedding for searchword, *wanted*, shape: (50,)

In [149...]
# apply PCA to all sentence embeddings
ncomp = 25
pca_ncomp = PCA(n_components=ncomp)
pca_ncomp.fit(embeddings_sentence_train)
embeddings_sentence_train_pca = pca_ncomp.transform(embeddings_sentence_train)

# apply same PCA to searchword
embedding_serchrd_wanted_pca = pca_ncomp.transform(embedding_serchrd_wanted[np.newaxis,:])
embedding_serchrd_wanted_pca = np.squeeze(embedding_serchrd_wanted_pca)

print("Shape of transformed embeddings : ", embeddings_sentence_train_pca.shape)
print('Embedding for searchword, *wanted*, shape:', embedding_serchrd_wanted_pca.shape)
print("Proportion of variance retained by transformed embeddings:", sum(pca_ncomp.explained_variance_ratio_))

Shape of transformed embeddings : (369930, 25)
Embedding for searchword, *wanted*, shape: (25,)
Proportion of variance retained by transformed embeddings: 0.8820319901119548
```

Filtering Data for Clustering

We will focus on a subset of the data for the initial clustering. The hope is that this subset will be much easier to cluster, if it is successful then the dataset could be widened at a later date.

The subset is those sentences including 'want', 'wants', 'wanted' and where the sentence is short, <24 tokens. This should help the ThreeCosMul function find a meaningful distance between sentences whilst using a variation of 'want' as the analogy word in all cases. It has been noted in an earlier chunk that the embedding for 'desire' and 'wish' are substantially different to that for 'want'. Attempting to cluster across all searchwords may be too ambitious in the first instance.

```
In [150...]
sentences_train['embedding_pca'] = [embed for embed in embeddings_sentence_train_pca.tolist()]
sentences_train['text_idxs_len'] = [len(text_idxs) for text_idxs in sentences_train['text_idxs']]

# 24 tokens is actually the 70th percentile
# Note, the tokens relate to BERT, but are a convenient measure of sentence length even if not used by GloVe
len_limit = int(sentences_train['text_idxs_len'].quantile([0.70]).iloc[0])
print("Sentence token length limit = ", len_limit)

query_text = "(text_idxs_len <= "+str(len_limit)+" and (searchwords_1st.isin(['want', 'wants', 'wanted'])))"
cluster_candidates = sentences_train.query().copy(query_text)
cluster_candidates.to_csv(os.path.join(path, 'cluster_candidates.csv'), index=False, sep='~')
print(len(cluster_candidates))

Sentence token length limit = 24
183950
```

Optimisation

We have n=184k sentences to analyse. This poses a number of problems because it leads to $n/2*(n-1)$ possible combinations of sentence pairs. Since n=184k then we have approx 17 billion distances to calculate! Such a distance matrix could not be held in memory for analysis by any clustering tool, or even be precalculated in any feasible amount of time, at least not with the equipment available for this analysis.

Therefore, we will submit the ThreeCosMul distance function as a custom function to the clustering tool. HDBSCAN is optimised to sample distances using this function. It greatly helps that we have already decided to use the embedding for the same searchword in all cases, ie 'wanted'. This avoids look-ups within the distance function, which slow the process.

The distance function will be executed at least millions of times, if not billions, it needs to be as efficient as possible. Therefore, the 'numba' package is used for precompilation of the function.

In [151...]

```
# New distance function, customs function submitted to HDBSCAN for clustering and UMAP for charting.
# same as last distance function, but default to embedding for 'wanted' for both searchword_a and searchword_b.
# therefore, submit only sentence embeddings, and get distance.
# NOTE: We use the numba package to pre-compile code, accelerates clustering by a factor of 10x !!

# These two functions are as per Levy and Goldberg 2014, 3CosMul
# Except, the division in 3CosMul is inverted so we get a measure of distance, not similarity
# 1 is then subtracted from that distance, so that two point at the same location have distance=0, not distance=1
@numba.jit(nopython=True)
def cossim_nb(x, y):
    x = x.astype(np.float32)
    y = y.astype(np.float32)
    numerator = np.dot(x,y)
    denominator = np.linalg.norm(x) * np.linalg.norm(y)
    return numerator / denominator

# precompiling cossim_nb() but not ThreeCosMul() appears to work best with HDBSCAN
# This is possibly due overhead, most of the work is done in cossim hence most benefit in precompiling it
# Whereas ThreeCosMul_nb() is just calling cossim, so the overhead of using precompiled function outweighs benefit
def ThreeCosMul_nb(sentence_a, sentence_b, serchwrld=embedding_serchwrld_wanted_pca, epsilon=math.pow(10, -6)):
    # epsilon simply prevents div by zero
    numerator = cossim_nb(sentence_b, serchwrld)
    denominator = cossim_nb(sentence_b, serchwrld) * cossim_nb(sentence_b, sentence_a) + epsilon
    # Note, this distance can return -ve distances, but so can cosine dist when applied to GloVe embeddings,
    # cosine is a standard function in HDBSCAN.
    # If you attempt to return +ve only distances, say by using abs(), then no clusters are found
    return (numerator / denominator)-1.0

# Later in this notebook we use the UMAP package
# UMAP needs ThreeCosMul to be precompiled...
@numba.jit(nopython=True)
def ThreeCosMul_ns(sentence_a, sentence_b, serchwrld=embedding_serchwrld_wanted_pca, epsilon=math.pow(10, -6)):
    numerator = cossim_nb(sentence_b, serchwrld)
    denominator = cossim_nb(sentence_b, serchwrld) * cossim_nb(sentence_b, sentence_a) + epsilon
    return (numerator / denominator)-1.0
```

HDBSCAN clustering

HDBSCAN has been chosen because it allows the data to speak for itself, it does not require us to set a number of clusters. However, we must specify min_cluster_size and min_samples. The previous analysis, using BERT embeddings, found the sentence embeddings clustered best when the min_cluster_size was 3000 and the min_samples was 1000.

Therefore, this arrangement will be chosen again, but also compared with alternatives to see whether the number of clusters found with this arrangement is indeed optimal. Optimal means the minimum number of clusters, i.e. simplest arrangement of the data, whilst not sacrificing the proportion of points clustered at all.

Read more on Hierarchical Density Based clustering at: <https://towardsdatascience.com/understanding-hdbscan-and-density-based-clustering-121dbbee1320e>

In [152...]

```
## HDBSCAN
cluster_candidates_embeddings_pca = np.array(cluster_candidates['embedding_pca'].tolist())

# for each HDBSCAN configuration...
cluster_configs = []
for config in [(1000, 750), (2000,1000), (3000, 1000), (4000,1500)]:

    # configure the scan
    min_cluster_size, min_samples = config
    hdbscanner = hdbscan.HDBSCAN(metric = 'pyfunc', func = ThreeCosMul_nb,
                                  min_cluster_size = min_cluster_size, min_samples = min_samples)

    # execute the scan
    start_time = time.time()
    hdbscanner.fit( cluster_candidates_embeddings_pca )
    end_time = time.time()

    # report results
    filename = 'clusters_'+str(ncomp)+D+'\
               _minclustersize_'+str(min_cluster_size)+\
               '_minsamples_'+str(min_samples)+\
               '_want_len'+str(len_limit)
    filename_pandas = os.path.join(path, filename+'.csv')
```

```

# save results, cluster assignment, probability of belonging to cluster, outlier_score, text and index of text
cluster_candidates['cluster'] = hdbscan.labels_
cluster_candidates['probability'] = hdbscan.probabilities_
cluster_candidates['outlier_score'] = hdbscan.outlier_scores_
cluster_candidates.to_csv(filename_pandas, sep='~')
cluster_configs.append((filename_pandas, min_cluster_size, min_samples))

# print progress
print(filename)
print(" Time taken to cluster :", str((end_time-start_time)/60), 'mins')

```

```

/home/oliver/anaconda3/envs/deep_gensim/lib/python3.8/site-packages/hdbscan/_hdbscan.py:1052: RuntimeWarning: invalid value encountered in double_scalars
clusters_25D_minclustersize_3000_minsamples_1000_want_len24
Time taken to cluster : 25.886362155278523 mins

```

Clustering Results

HDBSCAN finds 9-25 clusters, depending on configuration. It places approximately three quarters of the population into those clusters, so only a minority are not placed in a cluster. This is over 100k sentences placed in clusters, which is sufficient for deeper analysis of those clusters.

The optimal configuration appears to be min_cluster_size = 3000 and min_samples = 1000, same as with BERT embeddings. This returns the fewest clusters but includes as many points in clusters as any other arrangement.

In [153...]

```

# display summary of clusters

cluster_configs = [('clusters_25D_minclustersize_1000_minsamples_750_want_len24.csv', 1000, 750),
                   ('clusters_25D_minclustersize_2000_minsamples_1000_want_len24.csv', 2000, 1000),
                   ('clusters_25D_minclustersize_3000_minsamples_1000_want_len24.csv', 3000, 1000),
                   ('clusters_25D_minclustersize_4000_minsamples_1500_want_len24.csv', 4000, 1500)]

for config in cluster_configs:
    filename, min_cluster_size, min_samples = config
    cluster_candidates = pd.read_csv(os.path.join(path, filename), sep='~')

    # calculate summary of clusters
    cluster, count = np.unique(cluster_candidates['cluster'], return_counts=True)
    clusters_summary = pd.DataFrame({'cluster':cluster, 'count':count})

    print(filename)
    print(" Number of clusters : ", len(clusters_summary)-1)
    print(" Proportion clustered : ", sum(clusters_summary.query('cluster!= -1')['count']) / sum(clusters_summary['count']) )

clusters_25D_minclustersize_1000_minsamples_750_want_len24.csv
Number of clusters : 23
Proportion clustered : 0.7484805653710247
clusters_25D_minclustersize_2000_minsamples_1000_want_len24.csv
Number of clusters : 18
Proportion clustered : 0.6332536015221527
clusters_25D_minclustersize_3000_minsamples_1000_want_len24.csv
Number of clusters : 13
Proportion clustered : 0.7290568089154662
clusters_25D_minclustersize_4000_minsamples_1500_want_len24.csv
Number of clusters : 9
Proportion clustered : 0.6585648273987497

```

Build an Understanding of Each Cluster

The above pattern invites us to select min_cluster_size=4000, 66% of points are included, which gives 9 clusters. This is a useful number to investigate further. We need to understand whether these clusters are meaningful in anyway.

We will summarise the clusters, firstly by removing outliers, using HDBSCAN's outlier_score, then sorting them by the probability they belong to the cluster. This is a measure provided by HDBSCAN.

If we had been using a clustering tool like KNN then it would have been reasonable to find the centroid of the embeddings in each cluster and then measure the distance of each point to that cluster. However, the 3CosMul distance measure does not easily map to euclidean or manhattan distance and the clusters may substantially overlap in embedding space. So centroids are not favoured for finding representative points of hierarchical clusters.

Then we will attempt to view the clusters in 2D, by projecting the embeddings down to two dimensions using UMAP and LDA. UMAP has no preconception of the cluster assignments, it simply attempts to co-locate points which are nearby, using a distance measure, 3cosmul in this case. Whereas LDA is provided with the cluster assignments and tries to locate points according to both their locations and their clusters. Both give a visualisaiton, but it is particularly rewarding if UMAP's arrangement of the points agrees with the cluster assignments.

In [232...]

```

## Step 1 to find 'typical' examples of each cluster
# filter on points which are NOT outliers
minclustersize = 3000

```

```

minsamples      = 1000
filename = 'clusters_25D_minclustersize_'+str(minclustersize) +'_minsamples_'+str(minsamples) +'_want_len24.csv'
cluster_candidates = pd.read_csv(os.path.join(path, filename), sep='~')

# as we have just loaded the mebeddings in a pandas dataframe, must convert string to list
cluster_candidates['embedding_pca'] = [literal_eval(cell) for cell in cluster_candidates['embedding_pca']]

# There are two measures of affinity to a hierarchical cluster; probaility and outlier_score
# Get the lowest 10% quantile for outlier score and highest 10% quantile for the probability score,
# Then these are the most representative points.
outlier_quantile = pd.DataFrame(cluster_candidates.groupby(['cluster'])['outlier_score'].quantile(0.9)
                                ).reset_index(drop=False)

outlier_quantile.columns = ['cluster', 'outlier_quantile']
probab_quantile = pd.DataFrame(cluster_candidates.groupby(['cluster'])['probability'].quantile(0.5)
                                ).reset_index(drop=False)

probab_quantile.columns = ['cluster', 'probab_quantile']

quantiles = outlier_quantile.join(probab_quantile.set_index('cluster'), on='cluster')
cluster_candidates = cluster_candidates.join(quantiles.set_index('cluster'), on='cluster', rsuffix='_')

# filter down on the most representative points
query_text = "(outlier_score <= outlier_quantile) and (probability >= probab_quantile)"
cluster_neighbours_summary = cluster_candidates.query(query_text)
cluster_neighbours_summary = cluster_neighbours_summary[['cluster', 'text']] \
    .groupby(['cluster']) \
    .count() \
    .reset_index(drop=False)
cluster_neighbours_summary.columns = ['cluster', 'count_representatives']

# display new counts, after outliers removed
display(Markdown("##Summary of clusters after removal of outliers"))
display(cluster_neighbours_summary)

```

Summary of clusters after removal of outliers

cluster	count_representatives
0	44934
1	3001
2	3138
3	11059
4	5178
5	3687
6	5469
7	3000
8	16241
9	3909
10	5967
11	3000
12	22581
13	7929

The above results, where outliers are filtered out, gives us a new problem. There remain tens of thousands of examples in each cluster, far too many to view with the hope of understanding the exemplars for each cluster. With regret, we need to use the cluster centroid method to find representative examples. It may be that this is little better than random sampling, although that would be acceptable.

```

In [234...]
# Centroids are median points, so we need a median for each cluster
# The medians should exclude the outliers.
median_list = []
for row in cluster_neighbours_summary.itertuples():

    # apply filters, eg not an outlier and hi probability of being in cluster
    query_text = "(outlier_score <= outlier_quantile) " + \
                "and (probability >= probab_quantile) " + \
                "and (cluster=="+str(row.cluster)+")"
    cluster_embeds = cluster_candidates.query(query_text)['embedding_pca'].tolist()
    # NB, for distances in high dimensions we seek median not mean
    cluster_median = np.median(cluster_embeds, axis=0)

    # record the median
    median_list.append(cluster_median)

```

```

cluster_neighbours_summary['cluster_median'] = median_list
# the below row gets the median for each embedding's cluster
# The cluster's median is a mathematical location, not a real point.
# So we cannot use simple distance function, such as cosine, as proxy for 3CosMul
cluster_candidates = cluster_candidates.join(cluster_neighbours_summary.set_index('cluster'), on='cluster', rsuffix='_')

# because we are using geometric centroids we cannot compute distance using ThreeCosMul
# the distance must represent the distance between the embeddings, so must be cityblock or possibly cosine
# here we use cosine because already know it is a fair approximation to ThreeCosMul
# If you change the distance method then the common meaning of the new exemplars changes a little
cluster_candidates['distance_to_median'] = [cosine(a,b) for a, b
                                             in zip(cluster_candidates['cluster_median'].tolist(),
                                                    cluster_candidates['embedding_pca'].tolist())]

cluster_candidates['distance_to_median_rank'] = cluster_candidates \
    .groupby('cluster')[['distance_to_median']] \
    .rank(method='first')

# save data so far
filename = 'clusters_25D_minclustersize_{}{}_minsamples_{}{}_want_len24_medians.csv'.format(minclustersize, minsamples)
cluster_candidates.to_csv(filename, sep='~')

# lets see our data...
display(cluster_candidates[['text', 'searchwords_1st', 'distance_to_median', 'distance_to_median_rank']].head(n=3))

```

			text	searchwords_1st	distance_to_median	distance_to_median_rank
0	it may be assumed that the cardinals wanted to continue their policy		wanted		0.822224	11310.0
1	who wanted product from the band for the Christmas season		wanted		0.782141	32325.0
2	"Elektra obviously wanted the 'pretty' Jim Morrison		wanted		0.338987	1776.0

In [156...]

```

# get a random sample of points for which we will seek nearest neighbours
n_neighbours = 10
rank_limit = 20.0

# Sample points closest to each cluster's median
cluster_examples = []
for row in cluster_neighbours_summary.ittertuples():
    # get top items (by dist to median) within cluster
    query_text = "(outlier_score <= outlier_quantile) and (probability >= probab_quantile)" + \
                 " and (distance_to_median_rank <= "+str(rank_limit)+" and (cluster=="+str(row.cluster)+"))"
    shortlist = cluster_candidates.query(query_text)
    # sample from within those top items, but no more than is available
    n_sample = min(n_neighbours, len(shortlist))
    shortlist_sample = shortlist.sample(n=n_sample)
    # save data to list
    cluster_examples.append(shortlist_sample)

# concatenate all samples into one dataframe
cluster_examples = pd.concat(cluster_examples)
print('Example of data')
display(cluster_examples[['cluster', 'text', 'probability', 'distance_to_median_rank']]
       .query('cluster!=1')
       .groupby('cluster')
       .sample(n=1)
       .sort_values('cluster'))

```

Example of data

	cluster		text	probability	distance_to_median_rank
48504	0	but I felt he would get better and I wanted to make something for him to come back to"		1.000000	2.0
131692	1	he stepped down as the opposition leader in the State Legislative Assembly as he no longer wanted to helm the PAC		1.000000	18.0
140822	2	They did not want to take direct control of local administration in villages for several reasons		1.000000	16.0
171808	3	who wanted King to consult with the board before he proceeded with the expansion of the airport and other projects		1.000000	4.0
17555	4	He said he wanted to spend time with his family and travel across the country with his band		1.000000	14.0
91101	5	Canadian general Alain Forand didn't want to comment on Leslie's claim		0.520304	13.0
147457	6	"If you want others to be happy		1.000000	15.0
181858	7	Allied officials wanted to appoint a non-Nazi to take over administration of the city		1.000000	15.0
173695	8	His house has been taken over by displaced people and nobody wants to tell him how to find his wife		1.000000	3.0
152711	9	After Maggie tells him that she wants him to be her friend		0.722838	8.0
75476	10	I just don't want it done		1.000000	15.0
71744	11	in opposition to private architects who wanted to be able to work on federal projects		1.000000	13.0

cluster		text	probability	distance_to_median_rank
97962	12	They wanted to expand their reach in the Massachusetts region to the city of Boston	1.000000	16.0

View examples from the centre of each cluster

In [182...]

```
# present the samples from each cluster
pd.set_option('display.width', 2000)
pd.set_option('display.max_colwidth', 2000)
display(Markdown("**Cluster 0. Hypothesis: Desire for independence**"),
cluster_examples.query("cluster==0")[['text']].sample(n=5))
```

Cluster 0. Hypothesis: Desire for independence

text

75544	I wanted to do things that would trip me out
122553	Even then after coming to power I said if you want to say something good speak out
103602	so she did not want to do it
10833	After this I could do anything I want
3116	I knew straight away that this was what I wanted to do next

In [181...]

```
display(Markdown("**Cluster 1. Hypothesis: Desire for group affiliation**"),
cluster_examples.query("cluster==1")[['text']].sample(n=5))
```

Cluster 1. Hypothesis: Desire for group affiliation

text

86837	Bainovi was quoted as saying he wanted to play in the Premier League
131692	he stepped down as the opposition leader in the State Legislative Assembly as he no longer wanted to helm the PAC
93014	In 1840s he joined Slovak national revival because he wanted to codify language of his nation
56459	President Wilson had wanted Tarbell in the official U
181781	the 32-year old Beck announced he wanted to return to the National Hockey League

In [184...]

```
display(Markdown("**Cluster 2. Hypothesis: Desire for control**"),
cluster_examples.query("cluster==2")[['text']].sample(n=5))
```

Cluster 2. Hypothesis: Desire for control

text

122833	They wanted the citizens in the area to have direct control over future development
100445	The British government wanted complete separation of the island from Chinese administration
57550	Boiarsky also wanted the governor to have the authority to transfer funds between departments in state agencies
167408	The main reason why the Allies wanted to postpone the landings was because of their lack of supplies in 1943
140822	They did not want to take direct control of local administration in villages for several reasons

In [187...]

```
display(Markdown("**Cluster 3. Hypothesis: Wishes negotiated between parties**"),
cluster_examples.query("cluster==3")[['text']].sample(n=5))
```

Cluster 3. Hypothesis: Wishes negotiated between parties

text

171808	who wanted King to consult with the board before he proceeded with the expansion of the airport and other projects
24518	despite challenges from several federal agencies who also wanted to use parts of the yard
129121	Planning Board of the City who did not want to dilute the retail existing and planned elsewhere in the renewal area
132053	who wanted to move the bulk of their services from Torp
35892	many citizens wanted the reservoirs built to help protect the city

In [188...]

```
display(Markdown("**Cluster 4. Hypothesis: Desire for emotional connection**"),
cluster_examples.query("cluster==4")[['text']].sample(n=5))
```

Cluster 4. Hypothesis: Desire for emotional connection

text	
17555	He said he wanted to spend time with his family and travel across the country with his band
124561	He said he just wanted to leave the country
84040	He explained: "[Ken] was the sort who wanted to have a wife who would stay at home
67925	she came back and told Rob she wanted to have another go at their marriage
107920	and asked his father who he wanted to live

```
In [193...]: display(Markdown("'''Cluster 5. Hypothesis: Desire for dis-association (opposite of affiliation)'''"), cluster_examples.query("cluster==5")[['text']].sample(n=5))
```

Cluster 5. Hypothesis: Desire for dis-association (opposite of affiliation)

text	
152680	and doesn't want Avery and Cadence to be on the road with Layla
127513	Scratch the Hershey's because I don't want to endorse that or anything"
91101	Canadian general Alain Forand didn't want to comment on Leslie's claim
49806	I also don't want to take away from the reader's imagination
179781	Dodgers owner Walter O'Malley wanted to exploit new revenue streams to capitalize on the rabid fans of the Dodgers

```
In [198...]: display(Markdown("'''Cluster 6. Hypothesis: Desires about other's perspectives'''"), cluster_examples.query("cluster==6")[['text']].sample(n=5))
```

Cluster 6. Hypothesis: Desires from other's perspectives

text	
32504	but those eyes would make you be whatever he wanted you to be
82158	and you have to let him do what he wants to do
67545	but then they want you to be someone else
66488	and I could get wreck if I want to
147457	"If you want others to be happy

```
In [229...]: display(Markdown("'''Cluster 7. Hypothesis: Desire to impose authority over others'''"), cluster_examples.query("cluster==7")[['text']].sample(n=5))
```

Cluster 7. Hypothesis: Desire to impose authority over others

text	
181858	Allied officials wanted to appoint a non-Nazi to take over administration of the city
96949	This plan works for two weeks until the Nazi commander in their town wants to re-open the school
7655	who wanted to achieve control of the road networks in northern Luxembourg in order for their forces to move westward
127254	and passengers who wanted to make the connection had to leave the station
140608	Although the CIA wanted to intervene in a large scale

```
In [207...]: display(Markdown("'''Cluster 8. Hypothesis: Desire for emotional distance'''"), cluster_examples.query("cluster==8")[['text']].sample(n=5))
```

Cluster 8. Hypothesis: Desire for emotional distance

text	
122806	arguing that she only wanted the baby to be taken away
173695	His house has been taken over by displaced people and nobody wants to tell him how to find his wife
178307	and that he wanted his wife to be able to spend more time with their children
156652	but she was unable to decide what sort of work she wanted to do
111809	so she would not have been able to support her family in the way she wanted to

```
In [208...]: display(Markdown("'''Cluster 9. Hypothesis: Desire for emotional connection'''"),
```

```
cluster_examples.query("cluster==9")[['text']].sample(n=5)
```

Cluster 9. Hypothesis: Desire for emotional connection

text	
152711	After Maggie tells him that she wants him to be her friend
3292	who wants him to marry her; his friend Tom
139916	Jade told Big Brother she wanted to leave; she eventually changes her mind
5156	He realizes that he wants his girlfriend back and reunites with her
175415	She confesses that she wants to go to prom with him

In [213...]

```
display(Markdown("'''Cluster 10. Hypothesis: Obstinacy in relationships'''"),  
       cluster_examples.query("cluster==10")[['text']].sample(n=5))
```

Cluster 10. Hypothesis: Obstinacy in relationships

text	
159288	She said: "I don't want to say I'm retiring
75476	I just don't want it done
41893	saying "I don't want be like Hank
44086	"I want Elliot and Greg to be happy
134351	I don't want this done by just anyone

In [223...]

```
display(Markdown("'''Cluster 11. Hypothesis: Desire to impose authority over territory'''"),  
       cluster_examples.query("cluster==11")[['text']].sample(n=5))
```

Cluster 11. Hypothesis: Desire to impose authority over territory

text	
72925	There were some in Congress who wanted to repair the existing bridge
21056	The Americans wanted to establish military outposts and the Dakota wanted a new source of trading
27969	who wanted to direct the limited funds into construction of the Federal buildings
45424	wanted to have him oversee the removal of artworks from South Tyrol
97917	were about to open and the ministry wanted to award new concessions that included these airports

In [226...]

```
display(Markdown("'''Cluster 12. Hypothesis: Desire to win authority via allies'''"),  
       cluster_examples.query("cluster==12")[['text']].sample(n=5))
```

Cluster 12. Hypothesis: Desire to win authority via allies

text	
94447	They were able to discuss government opportunities and learn what the citizens needed and wanted from Saigon
159468	Many locals and some National Forest Service officials wanted to keep it closed to allow it to revegetate
97962	They wanted to expand their reach in the Massachusetts region to the city of Boston
178243	Some states may want to inspect and approve facilities
122755	to supply Gaza with electricity in exchange for the extradition of 17 wanted terrorists and other security demands

The above inspections are encouraging. Clusters appear to be mostly semantic, not synonym based, although there are clearly clusters focussed on large group identity (eg government) and those focussed on small group identity (family, friends). Notably all these clusters express a desire about relationships between people. None are about an object or external objective, all are between people. There will be more discussion of this later. First we'll complete the inspection of the clusters's strcutre by projecting the data down to 2D and then using topic modelling.

In [166...]

```
## Use UMAP to project data down to 2D, so we can plot clusters on 2D chart  
# apply umap only to clustered data, exclude non-clustered (ie group -1)  
cluster_candidates_nc = np.array(cluster_candidates.query("cluster!= -1")['embedding_pca'].tolist())  
  
for dist_met in [ThreeCosMul_ns, 'manhattan', 'cosine']:  
    # need embeddings in 2D, so project data down to 2 dims using UMAP  
    umap_reducer = umap.UMAP(n_components = 2, metric = dist_met,  
                            random_state = 0, n_neighbors = 30, min_dist = float(0.00))  
    cluster_candidates_umap2D = umap_reducer.fit_transform(cluster_candidates_nc)
```

```

#save results
if type(dist_met) != str:
    dist_met = dist_met.__name__

np.save(os.path.join(path, 'cluster_candidates_umap2D_'+dist_met+'.npy'), arr=cluster_candidates_umap2D)

```

/home/oliver/anaconda3/envs/deep_gensim/lib/python3.8/site-packages/umap/umap_.py:1727: UserWarning: custom distance metric does not return gradient; inverse_transform will be unavailable. To enable using inverse_transform method method, define a distance function that returns a tuple of (distance [float], gradient [np.array])

In [167...]

```

# Let's view those clusters
embeds_list = []
for dist_met in ['manhattan', 'cosine', 'ThreeCosMul_ns']:
    embeds = np.load(os.path.join(path, 'cluster_candidates_umap2D_'+dist_met+'.npy'))

    # convert to pandas
    embeds_pd = pd.DataFrame({ 'embed_x' : embeds[:,0],
                                'embed_y' : embeds[:,1],
                                'cluster' : cluster_candidates.query("cluster!= -1")['cluster'].tolist(),
                                'dist_met': [dist_met]*embeds.shape[0]})

    # filter out the outliers for the sake of having a reasonable plot scale
    x = embeds_pd.embed_x
    y = embeds_pd.embed_y
    is_inrange_x = x.between(x.quantile(.001), x.quantile(.999))
    is_inrange_y = y.between(y.quantile(.001), y.quantile(.999))
    clusters_umap2D_metric = embeds_pd[is_inrange_x & is_inrange_y]

    embeds_list.append(clusters_umap2D_metric)

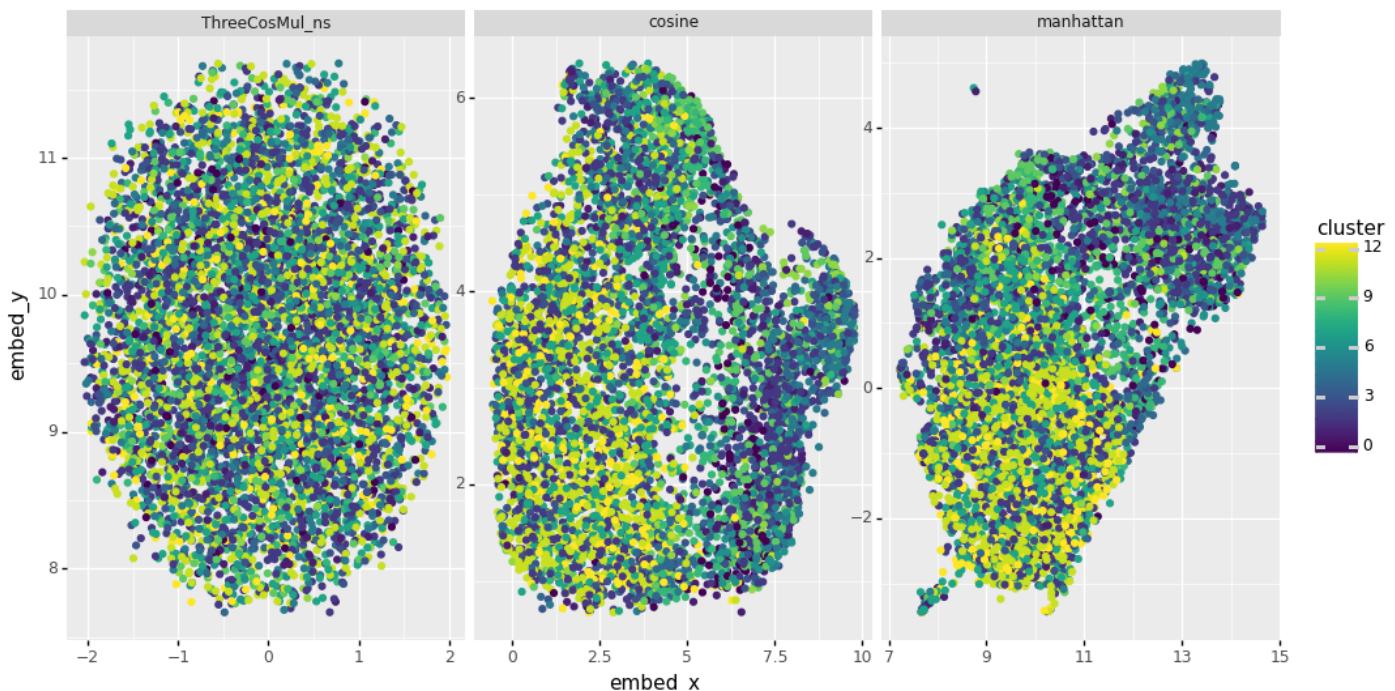
# Union of embeddings for all distance metrics, but filter out the points which were not clustered at all
clusters_umap2D = pd.concat(embeds_list).query("cluster!= -1")

# work on sample for plotting, else far too many points
clusters_umap2D_smp = clusters_umap2D.sample(n=20000)

# plot the data
(
    p9.ggplot(clusters_umap2D_smp)
    + p9.aes(x='embed_x', y='embed_y')
    + p9.geom_point(p9.aes(color='cluster'))
    + p9.facet_wrap(~dist_met, scales = 'free')
    + p9.theme.figure_size=(12, 6))
)

```

/home/oliver/anaconda3/envs/deep_gensim/lib/python3.8/site-packages/plotnine/facets/facet.py:390: PlotnineWarning: If you need more space for the x-axis tick text use ... + theme(subplots_adjust={'wspace': 0.25}). Choose an appropriate value for 'wspace'.



Out[167...]: <ggplot: (8738341876740)>

Clearly there is no relationship between embeddings and clusters when UMAP is tasked with preserving the ThreeCosMul distance. Some features are apparent when distance is measured by manhattan or cosine functions.

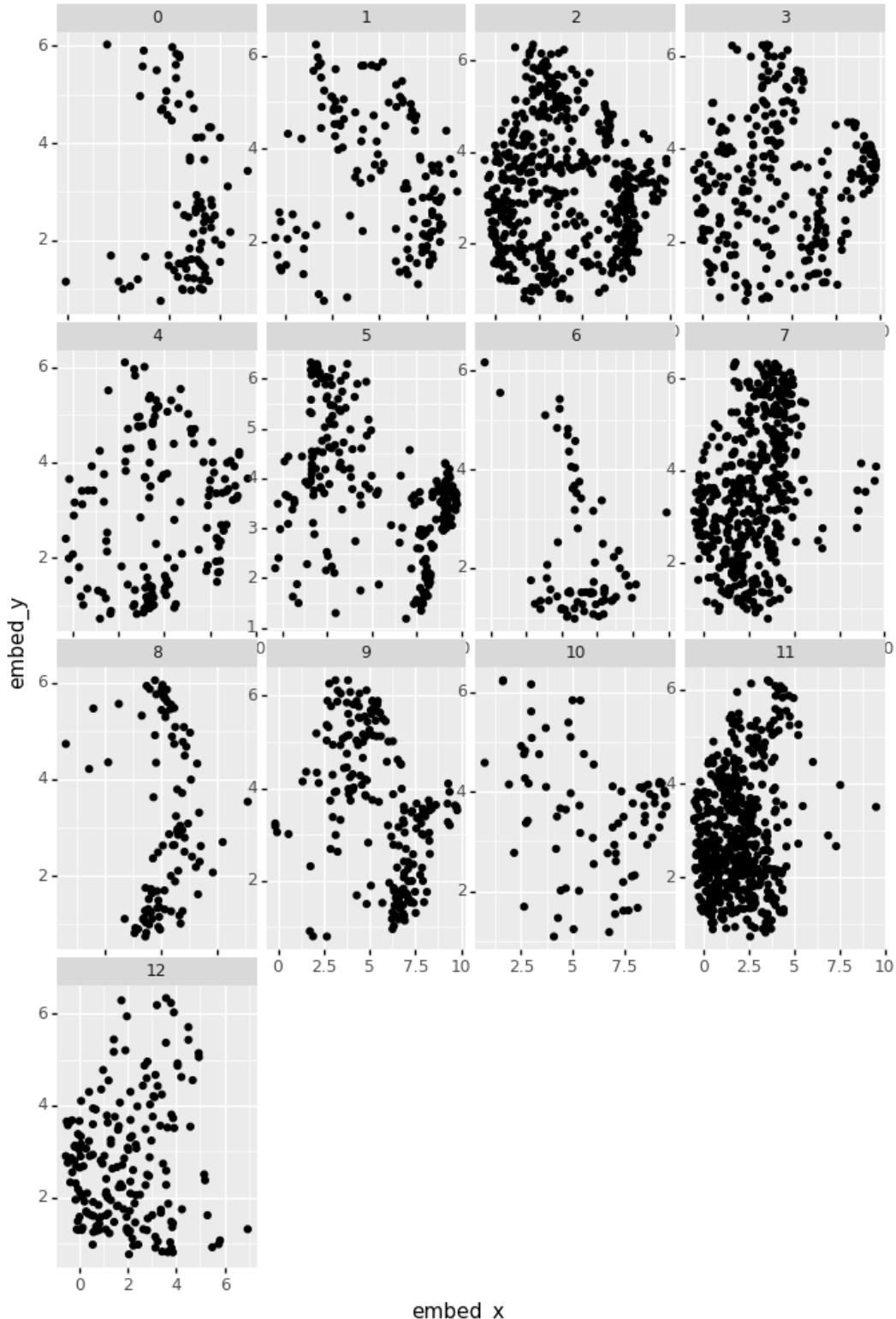
In the below code chunk those subtle differences are accentuated by printing one chart per cluster, based on UMAP preserving cosine distance. Nevertheless, the structure of each cluster exposed by UMAP remains weak.

In [168...]

```
# work on sample for plotting, else far too many points
clusters_umap2D_smp = clusters_umap2D.query("dist_met=='cosine'").sample(n=3000)

# plot the data
(
    p9.ggplot(clusters_umap2D_smp)
    + p9.aes(x='embed_x', y='embed_y')
    + p9.geom_point()
    + p9.facet_wrap('~cluster', scales = 'free')
    + p9.theme.figure_size=(8, 12))
)
```

/home/oliver/anaconda3/envs/deep_gensim/lib/python3.8/site-packages/plotnine/facets/facet.py:390: PlotnineWarning: If you need more space for the x-axis tick text use ... + theme(subplots_adjust={'wspace': 0.25}). Choose an appropriate value for 'wspace'.
/home/oliver/anaconda3/envs/deep_gensim/lib/python3.8/site-packages/plotnine/facets/facet.py:396: PlotnineWarning: If you need more space for the y-axis tick text use ... + theme(subplots_adjust={'hspace': 0.25}). Choose an appropriate value for 'hspace'

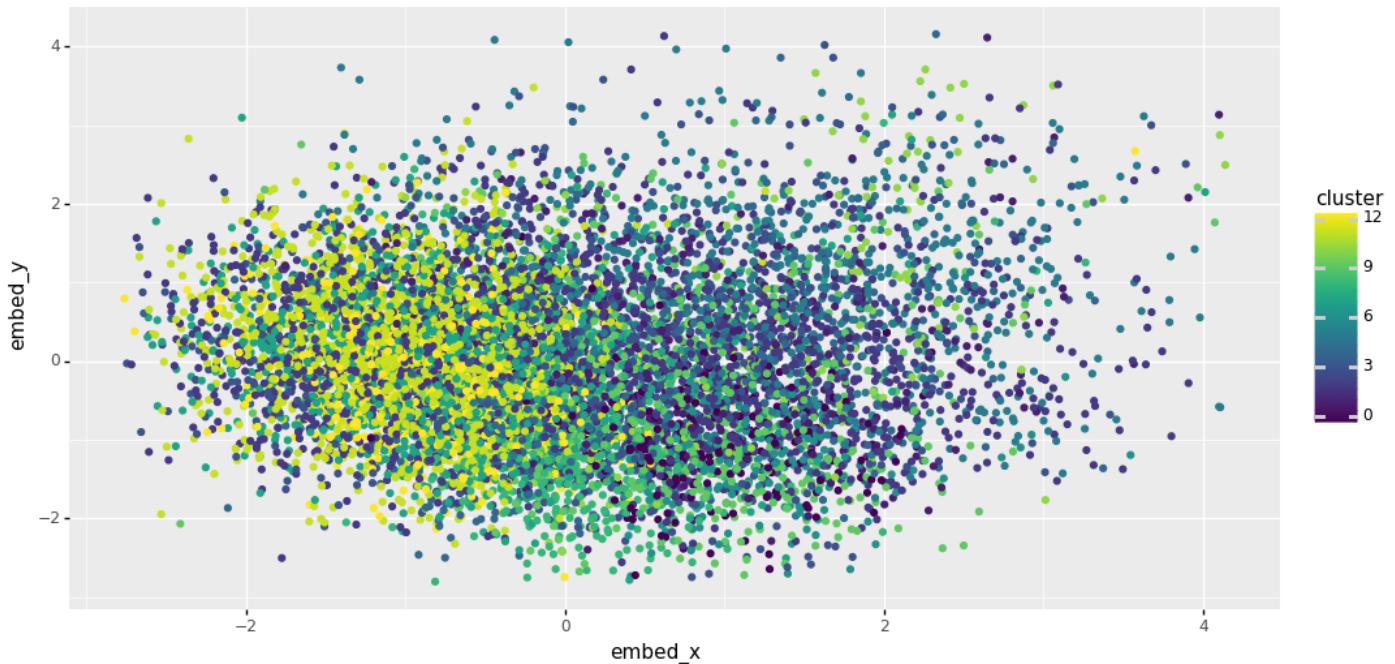


Out[168... <ggplot: (8738349092690)>

A second visualisation tool may help. This time, we will use LDA, which is provided both the embeddings and the cluster labels. It has enough

information to forcefully separate embeddings into their assigned clusters, if at all possible.

```
In [169...]  
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA  
  
# prep data for LDA  
is_clustered = cluster_candidates['cluster'] != -1  
LDA_embeds = np.array(cluster_candidates['embedding_pca'].tolist())[is_clustered,:]  
LDA_embeds = LDA_embeds * embedding_serchwrds_wanted_pca  
LDA_clusters = np.array(cluster_candidates['cluster'][is_clustered])  
  
# apply LDA  
X_LDA = LDA(n_components=2).fit_transform(LDA_embeds, LDA_clusters)  
  
# prep data for plotting  
clusters_LDA = pd.DataFrame({'embed_x':X_LDA[:,0], 'embed_y':X_LDA[:,1], 'cluster':LDA_clusters})  
clusters_LDA_smp = clusters_LDA.sample(n=10000)  
  
# filter out the outliers for the sake of having a reasonable plot scale  
x = clusters_LDA_smp['embed_x']  
y = clusters_LDA_smp['embed_y']  
is_inrange_x = x.between(x.quantile(.001), x.quantile(.999))  
is_inrange_y = y.between(y.quantile(.001), y.quantile(.999))  
clusters_LDA_smp = clusters_LDA_smp[is_inrange_x & is_inrange_y]  
del x, y, is_inrange_x, is_inrange_y  
  
# plot the data  
(  
    p9.ggplot(clusters_LDA_smp)  
    + p9.aes(x='embed_x', y='embed_y')  
    + p9.geom_point(p9.aes(color='cluster'))  
    + p9.theme.figure_size=(12, 6))  
)
```



```
Out[169...]: <ggplot: (8738374812694)>
```

LDA struggles to separate out the clusters for display in 2D. Faint structures are visible, but surprisingly weak.

It is curious that we can clearly perceive the different cluster characteristics when presented examples in text. However, when attempting to separate clusters using their embeddings, hardly any structure is visible.

This may be because the embeddings are insufficient to describe the location of 'sentence a' relative to 'sentence b' using ThreeCosMul. That function requires a third element, the searchword's embedding. This implies some transformation of the sentence embeddings using the searchword's embedding prior to expecting meaningful patterns in the clusters.

Confirm clusters are not attribute based

We'll apply TF-IDF, the topic extraction tool, to see whether any cluster has a particular bias towards a topic. We are seeking clusters based on semantics, not synonyms. We found that using the transformer embeddings the clusters were indeed directed by synonyms, which is why we moved on to GloVe embeddings. It would be satisfying if the topic words for each cluster did not focus on a subject, eg music, art or politics.

```
In [170...]  
# unite the texts of each cluster into one document per cluster.  
docs_df = cluster_candidates.query("cluster != '-1'")[['text', 'cluster']]  
docs_df['text_id'] = range(len(docs_df))
```

```

docs_per_topic      = docs_df.groupby(['cluster'], as_index = False).agg({'text': ' '.join})

# view one example
docs_per_topic.iloc[2,:]

```

Out[170...]

```

cluster
2
text      We learned that if you really wanted a movie to come out the way you wanted it to The resident may want to disclose assets when they have not submitted a return on time " saying they wanted to convert the university into an "American" propaganda institution Charles III wanted to keep fighting the following year which state officials had already said they wanted to keep to compensate the state for building new garages for the teams Georgia during the time of his service in the Pacific show that he wanted to be "where things are happening It also wants various policies to make Venezuela more democratic I really want to listen to this song showing the viewers videos sent in by men who want to overcome their fear She was not particularly pretty but she fancied herself so and wanted everyone else to think so too The British general did not want to lay siege and withdrew his forces to Canada "I wanted some real party songs I have never yet been able to discover just what the Sinn Feiners want General Wrangel stayed on a quay until all who wanted to leave Russia had done so Once asked if he wanted to be remembered as a statesman Korg stated how Hulk wanted to be left alone and how he made allies that were like family to him "I've always wanted to go to Michigan The region wants to be more attractive on the conference market which would create more job opportunities outside the tourist season I want you to do an Indian dance in front of his box and wanted to change what she felt was an uninteresting image of women in Latin American cinema if doctors wanted to record an accurate brain temperature they wanted to appoint him as a member of the Academy or are wanted fugitives in their countries with the exception of persecution victims When I finished that album I never wanted to speak to him again I don't know if I would have wanted to see Elvis like that He wanted her to accompany the "Big Four" on the final nights in the House but I wanted my first solo recording to be a...
Name: 2, dtype: object

```

In [173...]

```

from sklearn.feature_extraction.text import CountVectorizer

# importance value, based on frequency, for each word in a cluster
def c_tf_idf(documents, m, ngram_range=(1, 1)):
    count = CountVectorizer(ngram_range=ngram_range, stop_words="english").fit(documents)
    t = count.transform(documents).toarray()
    w = t.sum(axis=1)
    tf = np.divide(t.T, w)
    sum_t = t.sum(axis=0)
    idf = np.log(np.divide(m, sum_t)).reshape(-1, 1)
    tf_idf = np.multiply(tf, idf)

    return tf_idf, count

tf_idf, count = c_tf_idf(docs_per_topic.text.values, m=len(cluster_candidates))

# score words within each cluster by their frequency
def extract_top_n_words_per_topic(tf_idf, count, docs_per_topic, n=20):
    words = count.get_feature_names()
    labels = list(docs_per_topic.cluster)
    tf_idf_T = tf_idf.T
    indices = tf_idf_T.argsort()[:, -n:]
    top_n_words = {label: [(words[j], tf_idf_T[i][j]) for j in indices[i][::-1] for i, label in enumerate(labels)]}
    return top_n_words

top_n_words = extract_top_n_words_per_topic(tf_idf, count, docs_per_topic, n=20)

# convert top_n_words to pandas for convenience
top_n_words_pd = []
for cluster in top_n_words:
    word_freq = pd.DataFrame(top_n_words[cluster])
    word_freq['cluster'] = [cluster]*len(word_freq)
    word_freq.columns = ['word', 'freq', 'cluster']
    top_n_words_pd.append(word_freq)

top_n_words_pd = pd.concat(top_n_words_pd)

# create mask to remove the top_n_words which are simply the searchwords we originally specified
searchwords = ['wanted', 'wants', 'want', 'desired', 'desires', 'desire', 'wished', 'wishes', 'wish']
mask = [word not in searchwords for word in top_n_words_pd['word']]

# print the top 12 words per cluster
for cluster in top_n_words:
    top_n_words_cluster = top_n_words_pd.loc[mask, :].query("cluster=="+str(cluster)).reset_index(drop=True)
    top_n_words_cluster = top_n_words_cluster['word'].tolist()
    top_n_words_cluster = top_n_words_cluster[0:12]
    if cluster > -1:
        print("Cluster ", cluster, ":", *top_n_words_cluster)

```

```

Cluster 0 : did does said just tells make people know says told come really
Cluster 1 : make just did song really life does said film love like play
Cluster 2 : did make people just said really like know music don new government
Cluster 3 : didn did don people doesn said just make new film tells know
Cluster 4 : don did people make didn just like said work time know doesn
Cluster 5 : didn don doesn really just like make know play band did said
Cluster 6 : does just make people did know things really say away tells life
Cluster 7 : did new make said government film work people use time family decided
Cluster 8 : did people life father family children help mother parents does child told
Cluster 9 : did know tells just people play make does like father really marry
Cluster 10 : doesn don didn said know like did make told does really tells
Cluster 11 : did new people use government make create said war city company time
Cluster 12 : did people new government use make war work said continue create time

```

Cluster Topic Results

The above analysis is somewhat encouraging. Topic words such as film, song, government, family, friend etc all arise. However, they are spread over a number of clusters. We don't have a cluster which monopolises music or film, for example. We cannot say all the clusters are agnostic to topic. Notably clusters 8 and 9 are family oriented, clusters 11 and 12 are government oriented, whereas cluster 7 is inbetween having elements of family and government. However, we can say that these clusters appear more agnostic than those produced by the transformer embeddings.

Robustness of Cluster Meaning vs Cluster Parameters

Cluster min_cluster_size

Reducing the min_cluster_size increases the number of clusters and therefore the granularity of the desire. For example, independence is an umbrella for a number of other desires which the clusterer brings together when min_cluster_size increases.

We can imagine how the above themes would be further aggregated by increasing min_cluster_size, clusters 11 and 12 on imposing or winning authority are clearly close. Further analysis could be conducted on how the themes aggregate as the min_cluster_size is increased from 1000 to 4000.

Sentence Embedding Formulation

The embeddings analysed here used simple averages of the GloVe word embeddings to form a sentence embedding. Discrete Cosine Transforms were attempted in a later execution of the above code. It was hoped that DCT would preserve some element of word sequence lost by simple averaging. However, the clusters formed around sequences, for example, there was a cluster for sentences beginning 'I don't want to...', which is a common start to a sentence. So, counter to expectations, simple averaging was more useful for finding semantic analogies.

PCA Compression Ratio

It has been convenient to cluster data in 25 dimensions as this is fast and allows investigation of the effect of clustering parameters. However, it is possible to cluster the data in the original 50 dimensions, or even 300 dimensions if sufficient computing power is available.

The above code was re-run but without PCA compression, rather leaving the embeddings at 50 dimensions. This results in clusters which adhere more to topic modelling expectations, which was the original problem with transformer based embeddings. For example, in 50dims there are clusters for 'want' in the context of business, military, music etc. Compression to 25 dimensions, and its associated loss of 11% of variance, appears to blur the topic granularity, possibly allowing clusters to form around themes common to subject areas not topics. This was a happy accident but the conjecture is worth further investigation by clustering with data in say 15 dims rather than 25. Do we obtain even less synonym oriented themes but more semantic themes?

Distance Measure for Distance to Cluster Centroid

There is no easy way to find exemplars in clusters identified by HDBSCAN. Many areas within a cluster may share the same density, so which should be considered an exemplar? This analysis has fallen back on using geometrical centroids, based on the embeddings. This appears to work but it is unsatisfactory as it assumes the clusters are globular. Yet the very advantage of HDBSCAN is that a cluster could be any shape, even annular, such that the centroid is not representative of the cluster. Projecting embeddings down to 2D indicates clusters are globular, not annular, as does the success of the cluster characterisation. Nevertheless, changing the measure of distance to the centroid, for example from cosine to manhattan (aka cityblock), subtly changes the nature of those characterisations. These precise clusters and their characterisations are not fixed in stone, convincing alternatives appear possible. However, the result that all clusters are about relationships remains robust between distance measures.

Searchword

The clustering exercise could be repeated for the other searchwords, but that is a task for future work. Of course, this work shows how other types of analogy could be investigated, for example how do other motivations cluster? Fear, hate, love? Plutchik's entire wheel of emotions could be investigated.

Corpus

The chosen corpus has been Wikipedia 2014. This is a narrative about our history, which suits the investigation of the hypothesis presented by Bertrand Russell. It would be fascinating to see how the clusters form when trained on a different corpus, say the book corpus, which is comprised of fictional works and language directed at providing a narrative of character's lives. Such as corpus is available in a format easily compatible with the above code:

Conclusion

The above clusters have achieved what the transformer clusters did not. They relate to themes which, to a large extent, cross subject boundaries. It is of particular interest that cluster centroids do not present classic 'desires', which would also capable of reaching across subjects. Such classic desires may include ambition, wealth, fame or power. Let's list the characterisations of the clusters that HDBSCAN actually found:

- 1 Hypothesis: Desire for independence
- 2 Hypothesis: Desire for group affiliation
- 3 Hypothesis: Desire for control
- 4 Hypothesis: Wishes negotiated between parties
- 5 Hypothesis: Desire for emotional connection
- 6 Hypothesis: Desire for dis-association (opposite of affiliation)
- 7 Hypothesis: Desires about other's perspectives
- 8 Hypothesis: Desire to impose authority over others
- 9 Hypothesis: Desire for emotional distance
- 10 Hypothesis: Desire for emotional connection
- 11 Hypothesis: Obstinacy in relationships
- 12 Hypothesis: Desire to impose authority over territory
- 13 Hypothesis: Desire to win authority via allies

These are, of course, just hypotheses formed from small selections of each cluster, near the centroid of points with highest probability of belonging to the cluster. If we could travel across a cluster and take samples at regular intervals then surely we would observe these central themes morphing into something more generic and then into the theme of a neighbouring cluster.

We can see the theme to the above themes. They are all desires defined by our relationships with each other, some clusters focus on our relationships in large groups (teams, governments), others focus on our relationships in small groups (friends, family). Unlike the classic desires they are not driven by objects or objectives, other than the objective of being relevant to each other.

This is a fascinating and even rather beautiful outcome, albeit constrained by the above queries on robustness.

Other Implications

Training Transformers for Semantic Analogies

The joy of transformers is that they can easily be 'transfer trained' for new uses. Could we transfer train a transformer for sentence prediction on semantic analogies, perhaps by inputting the first sentence in a semantic pair and training it to give a semantically similar sentence as an output? Collecting the data for doing this was previously very awkward, there is no corpus of semantically analogous sentences. This work indicates a path to building such a corpus.

Would the sentence embeddings derived from such a transformer provide more structured clusters? Would they improve performance on the aspects of language which BERT based models are weak?

Generalising ThreeCosMul

The lesson of this analysis may not be the fundamental desires which drive people. There is some scope to tune clustering and get slightly different answers. Furthermore, there are limits to the robustness of the result, as discussed above.

Instead, an important lesson may be the effectiveness of ThreeCosMul in finding generalised patterns, independent of topic. ThreeCosMul was proposed by Levy and Goldstein to measure the analogy between words. This analysis applies it to find themes between a word and two sentence embeddings, a task which it performs well.

Could its usefulness be extended yet further? Could a similar measure be applied to find common patterns across any other features which can be reduced to a vector? Not forgetting the role of compression in blurring out features aligned to specific topics rather than general semantics.

Progress in finding generalised relationships may be found via a distance function which requires input from a third vector, which is the feature we are generalising for. Almost all other distance or cost functions demand only the vectors a and b , giving no place to a third vector, no clue to the common theme being sought. This is analogous to human perception, which is always coloured by our preconceptions, those are the features we are testing the scene for, either consciously or not. Machine Learning is powerful when agnostic, neutral to the outcome, but when seeking to better model people we may need a preconception, a third vector to test for.

DELETE CHARGEABLE AZUREML WORKSPACE

In []:

```
# Delete resources in AzureML
```

```
# Resources will continue to incur charges unless deleted, Look out for those SSD storage charges, ouch!
ws.delete(delete_dependent_resources=True)
```