Domain Classifier

Throughout the internship I met most Wednesdays with my supervisor. During this time we had conversations that ranged from the scope of the project to ways to approach particular solutions. Near the end of my work on the back-translation verifier, we started discussing the idea of a domain classifier. Domain classification is the process of assigning documents to various topics, which are generally semantically linked (Aponyi, 2021). In this case ILAD stated the following: "The goal for the domain classifier is to gather 4,800 total hand-labeled sentences for the following domains. "

1. story/narrative- 1.2k
2. culture/tourism- 1.2k
3. History- 1.2k
4. Instructions-1.2k

As with the rest of my internship, I worked with Indonesian data with the intent to apply any tools created to Alas eventually, as well as potentially leveraging the relationship between Indonesian and Alas such that Indonesian data could be used to inform investigation into Alas. Considering this, it wasn't my task to find these 4800 sentences, but rather to build a tool that could apply in a variety of language contexts by receiving sentences and classifying those sentences into various domains. As a result, my supervisor and I agreed that it might be interesting to let a language agnostic system identify its own domains, which could then be labeled after the fact by a human.

Of all the parts of the project, this is the one that evolved the most over time. Perhaps that is due to the fact that more actual time passed--I worked on it from approximately September until December while also juggling classes and teaching high school--or perhaps it is due to the nature of the project itself. More than in previous portions of the internship, I found myself stuck at certain points, and retrofitting my as-yet-incomplete code. What I ended up with in the end is, I think, superior to what I initially imagined, but I of course wish that I could have saved time by simply starting with the more robust ideas.

I used all of the language model that I had built for the back-translation verifier as a starting point, and from there I produced code that would iteratively sift through words, grouping the words that are the least dissimilar into a 'domain'. The code is demonstrated below. The function works on a recursive sifter. The way it works is by receiving domains. On the first call the domains are simply dictionaries of random words--the most frequent words being the first, as my intuition is that high frequency lexical words will have high domain impact. Then the sifter checks all of the words against every other word in its domain using the `ptable` variable, which had previously calculated the similarity between every pair of words within all domains as defined by vectors and cosine distance. The resulting orders them from most similar to least similar. The most similar word is essentially the most 'fitting' word for that domain. Then x, where x is defined by `siftcount` words are removed from the domain and randomly reassigned to another domain. The sifter then calls itself recursively until one of two things eventually happens. Either a given amount of epochs concludes or a given amount of words have been sifted out. These sifted out words can then be fed back into a future batch of words.

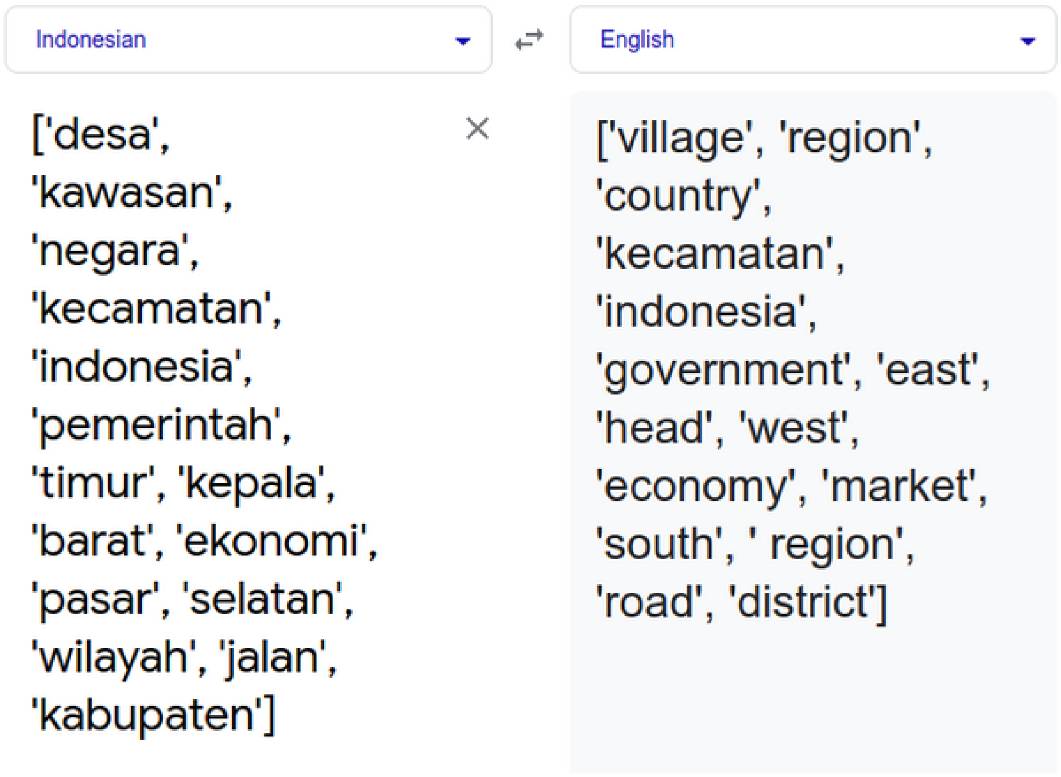
Note that the parameter `domain` is a class that included such variables as `children` ,`vector`, `name`, `integrity`, among others. This `domain` class is an abstract data class which had the capacity to link together with similar domains in a parent-child relationship.

```
def sifter(self, domain, ptable, context, epochs: int, siftcount: int,
quantity, dtol: int):
    # Recursive, two ways to escape, either max epochs reached, or max
siftcount reached
    epochs -= 1
    if epochs == 0 or siftcount == 0:
        return domain
    else:
        dprobs = {}
        for k, v1 in domain.children.items():
            v1.age += 1
            wprob = []
            for w1 in v1.vector:
                sim = 0
                for w2 in v1.vector:
                    if w1 == w2: continue
                    #Finding the distance (according to vectors) between two
words.
                    sim += ptable[w1][w2]
                wprob.append((w1, sim))
            dprobs[k] = wprob

        # Get the similarities for every word as referenced against every other
word in its domain.
        # Lowest similarities will be sifted out and placed in another domain.
        reorder = list(domain.children.keys())
        random.shuffle(reorder)
        for k, v in dprobs.items():
            v.sort(key=lambda x: x[1], reverse=True)
            total = sum([val[1] for val in v])
            self.dtotals.append((k, total, 'epoch', epochs))
            for worst in v[-siftcount:]:
                domain.children[k].vector.remove(worst[0])
                popped = reorder.pop(0)
                reorder.append(popped)
                if worst[0] in self.rejects:
                    self.rejects[worst[0]] += 1
                else:
                    self.rejects[worst[0]] = 1

                if self.rejects[worst[0]] < 20:
                    domain.children[reorder[0]].vector.append(worst[0])
                else:
                # This is the way we drop words that aren't part of any of the
x domain
                    self.gcounter += 1
                    if self.gcounter == dtol:
                        self.gcounter = 0
                        siftcount -= 1 #as words drop sifting gets finer
                    if siftcount == 0:
                    #if too many words are dropped, then it goes to a balancer
function before returning
                        domain = self.balancer(domain, ptable,
len(domain.children))
                        #used to avoid unbalanced group sizes
                        return domain
        totalprob = sum([val[1] for val in self.dtotals[-quantity:]])
        wordcount = sum([len(v.vector) for v in domain.children.values()])
        self.totprob.append(("totalprob:", totalprob, "avgwrdprob", totalprob /
wordcount, "totalwordcount", wordcount))
        domain = self.sifter(domain, ptable, context, epochs, siftcount,
quantity, dtol) # recursive call
    return domain
```

Early usages of this sifter were promising, a quick pass through Google translate shows the following.

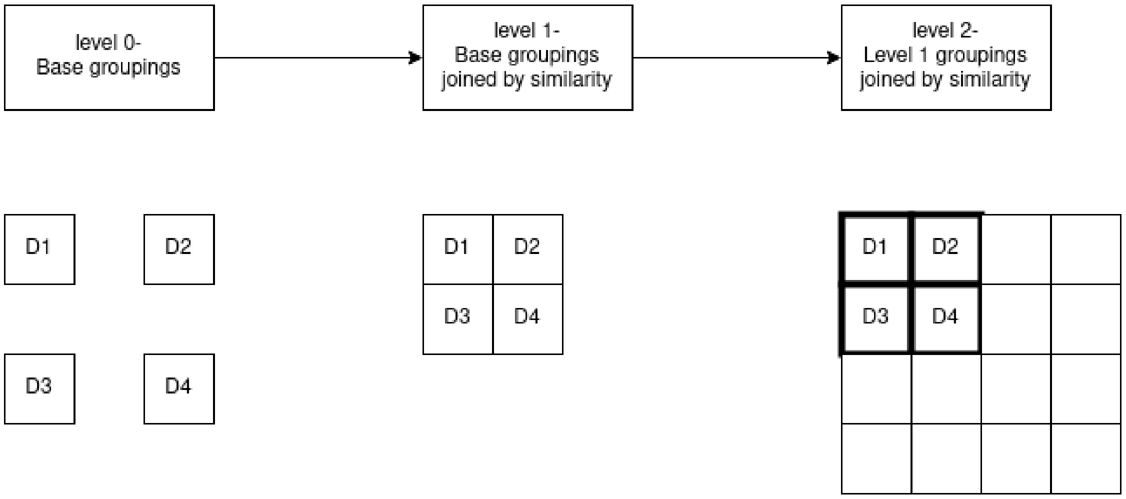


['desa', 'kawasan', 'negara', 'kecamatan', 'indonesia', 'pemerintah', 'timur', 'kepala', 'barat', 'ekonomi', 'pasar', 'selatan', 'wilayah', 'jalan', 'kabupaten']

['village', 'region', 'country', 'kecamatan', 'indonesia', 'government', 'east', 'head', 'west', 'economy', 'market', 'south', ' region', 'road', 'district']

My goal was to reproduce this effect across at least 50,000 words, and then apply it to documents, or in the case of my corpus, sentences. The underlying concept here is similar to KNN, K-Nearest Neighbors. The idea is that words are defined by their neighbors. However, KNN is a supervised learning technique, and I am designing a model for unsupervised learning. I think much of the need for evolution in my thinking is a result of this exact issue. What is appealing about KNN is the idea of finding the nearest neighbors, which can be accomplished through the vectors, every unique word being a dimension. Thus words such as `desa` and `kawasan` have a lower distance than say `desa` and `abednego` (a real example drawn from processed data--I can only assume I have a few sentences in the corpus relating to the biblical story, as the vector includes also `sadrakh` and `mesakh`). This reveals a challenge: there are no absolute boundaries against which words can be compared--or to put it in other words, the domains found are highly susceptible to the nature of the corpus. If there are sufficient sentences present, a major domain might be `biblical`.

Scalability is again an issue--here on multiple levels. On one hand, there is the simple fact that repeatedly comparing every word against every other word is computationally untenable. Therefore as domains grow, it becomes impossible to sift words in the same fashion. My solution to this issue was to build these groupings of approximately 20 words across the entire corpus in batches. This was counter-intuitive to me at first, because it felt far from optimized, as it effectively means that many words will not be compared against each other. Taking the above example, `region` can only be in the grouping with `village` if it is in the same batch. However, in building domain classifiers, groups of 20 words are trivial in size. As these groupings grow, the identity of the words begins to become less and less discernible in any case. What is necessary is a system that merges groupings, while also remembering the smaller groupings. This can be illustrated with the following:



These levels can be parsed either on the smallest level, or seen as bigger chunks. Thus the level one grid of four tiles would be a grouping of approximately 80 words, and the level 2 would be a grouping of approximately 320 words. It is of course, possible, that groupings would be more or less than four; here four is used for illustrative purposes. In practice this can be accomplished by the way the data is saved. Below is a view of eight curated words from one of the tables in a database. `Abednego` and `desa` are dissimilar words, and that dissimilarity can be stated by the fact that they are never in the same grouping even at the highest level, which entails thousands of words. On the other hand `abednego` and `almond` are fairly similar, not being in the same level 0 grouping, but being in the same level 1 grouping. Finally `desa` and `mobil` are as similar as possible, being even in the same level 0 domain, which is a result of being processed in the same batch and being sifted into the same grouping. Note that each descending level shows its own identity, indicated by the right most number, as well as all parent domains, the leftmost digit indicating the highest/largest domain within which the word resides.

word	level4	level3	level2	level1	level0
abednego	2	2.2	2.2.1	2.2.1.3	2.2.1.3.2
akuarium	1	1.4	1.4.4	1.4.4.3	1.4.4.3.2
alarm	1	1.2	1.2.4	1.2.4.4	1.2.4.4.2
almond	2	2.2	2.2.1	2.2.1.3	2.2.1.3.4
banjar	1	1.2	1.2.1	1.2.1.1	1.2.1.1.4
basket	1	1.2	1.2.1	1.2.1.3	1.2.1.3.3
desa	1	1.2	1.2.1	1.2.1.1	1.2.1.1.1
mobil	1	1.2	1.2.1	1.2.1.1	1.2.1.1.1

This system is built to be dynamic, such that the amount of levels is not predetermined. Thus a larger corpus will have more levels, but each word can be easily identified, and called up at any given level, essentially expressing radiating relationships. Note that `desa` and `mobil` share a level 0 relationship; then `banjar` shares a level 1 relationship with each of these; then `basket` shares a level 2 relationship, and `alarm` shares a level 3 relationship, and finally `akuarium` shares a level 4 relationship. This design creates a few assumptions, namely that on some level all words are related, and that these levels can be stratified. Additionally distance can be measured as the smallest level wherein the words share a domain.

Here we have almost reached domain classification; however, one critical element is missing: a functional identity. If every domain at every level had a unique and meaningful name, then these names could be used as computer driven domains upon which a user could classify documents. However, there are two main issues with this. The primary issue is that the task of organizing words is very different from the task of selecting the most meaningful or impactful word in a grouping. One might consider selecting the centroid of any grouping--the least different word from all the other words--but in practice this does not line up with human intuition. The word `noun` classifies a great many words, while in practice--except for in the company of linguists--it is rarely visible and highly unlikely to be chosen as the centroid of all nouns.

The secondary issue is that of which level to use for domain classification. If the user wants only four domains, then it is likely that such a user will want a level 4 or higher domain. If, however, the user is seeking out 12 domains, then perhaps level 2 or 3 is more appropriate. Ultimately from a design standpoint, I want the user to be able to dynamically select these attributes, and again, ultimately be able to apply this to documents.

To solve this I realized that I needed to use the relational aspect of databases, such that the above table, which I called the `domainicon` where the primary key is the `word` can be related to the `keystones` table, in which the primary key is the `domain` itself. Critically, the keystones table is populated with every domain at every level, such that there is a unique entry for 1, 1.2., 1.2.1, 1.2.1.1, and so on. The other columns in this table can be a combination of human targeted or computer suggested topics, that are represented as the log value of the target words divided by all words of that domain level.

domain	level	pemerintah	sejarah	pusat	struktur	obat	wilayah	pelaku	ekonomi
1	4	0.00161377	0.00121057	0.00282239	0.00282239	0.00282239	0.00282239	0.00241968	0.00268817
1.2	3	0.00171282	0.000571265	0	0	0	0	0.0102332	0
1.2.1	2	0.00496279	0.00248447	0	0	0	0	0.0438026	0
1.2.1.1	1	0.0222231	0	0	0	0	0	0.184192	0
1.2.1.1.1	0	0.0540672	0	0	0	0	0	0.693147	0

Note that the first two target columns `pemerintah` , which means government, and `sejarah`, which means history, are human created domain targets, that are directly written into the domain class infrastructure as demonstrated below. As the domains are formed the class checks itself to see if it has any of these target words, and then increases its count, which ultimately will be used to allow words not targeted to contribute towards document classification for a particular target domain by co-existing within the same domain.

```
self.dtargets = [("pemerintah", ["pemerintah", "walikota", "kantor",
"presiden", "raja", "kekuasaan", "terpilih", "hukum", "istana", "suara",
"pemimpin", "pajak", "negara"]), ("sejarah", ["sejarah", "masa lalu", "cerita",
"peristiwa", "ingat", "ceritakan kembali", "tulis", "penting", "tragedi",
"rakyat", "akun", "dokumen", "budaya ", "agama", "hari", "perubahan"])]
```

For example, though `banjar` is not a human targeted word, it can still contribute toward the government domain because it co-exists with at least one target word at the domain level of `1.2.1`, and will, of course co-exist at the level of `1` and `1.2` as well. It is also possible that a word such as `banjar` would contribute towards other target domains, as it does here: `sejarah` and `pelaku`. This conforms to my desire for the program, and is reflective of reality. A word meaning neighborhood, might be relating to governance, history, or crime, depending on the context. Which way the document is ultimately classified, is the aggregate of these scores across all words that exist in the `domainicon` table as referenced against the `keystones` table.

Once these two tables are populated, it takes surprisingly little code to query the database and return a dictionary of every target (columns from the keystone table), across an entire document. The code is demonstrated below. The parameter `target` is simply a string. Any words that are absent from the database will simply not contribute towards domain classification.

```
def classifier(self, target):
    query = "DESCRIBE keystones;"
    self.repo.cursor.execute(query)
    dt = list(self.repo.cursor)
    totals = {}
    mapping = {}
    for i, t in enumerate(dt[2:]):
        totals[t[0]] = []
        mapping[i] = t[0]
    target = target.lower()
    target = target.split()
    for t in target:
        query = f'SELECT * FROM domainicon WHERE word = "{t}";'
        self.repo.cursor.execute(query)
        levels = list(self.repo.cursor)
        if not levels: continue
        levels = list(levels[0])
        levels.reverse()
        for l in levels[:-1]:
            query = f'SELECT * FROM keystones WHERE domain = "{l}"'
            self.repo.cursor.execute(query)
            domains = list(self.repo.cursor)
            domains = domains[0]
            for i, d in enumerate(domains[2:]):
                totals[mapping[i]].append(d)
    for k, v in totals.items():
        totals[k] = sum(v)
    return totals
```

While I still want to refine the parameters I use across the domain classifier, as an example of the ultimate result, I ran the following sentence of Indonesian `"Kita harus mengingat dan menuliskan masa lalu agar tidak menjadi tragedi"` which translates in Google translate as "We have to remember and write down the past so it doesn't become a tragedy". While artificial in nature, it demonstrates the concept by identifying `sejarah` as the the most pronounced target domain.

```
{'pemerintah': 0.00161377, 'sejarah': 0.045030612000000005, 'pusat': 0.00282239,
'struktur': 0.00282239, 'obat': 0.00282239, 'wilayah': 0.00282239, 'pelaku':
0.00241968, 'ekonomi': 0.00268817, 'wanita': 0.00282239, 'pengunjung': 0.00255394,
'hotel': 0.00268817, 'prancis': 0.00335909, 'kalangan': 0.00268817, 'peran':
0.00295659, 'proyek': 0.00282239, 'tentara': 0.00295659, 'peristiwa': 0.00295659,
'new': 0.00268817, 'komunitas': 0.00309078, 'ulang': 0.00295659}
```

Perhaps what is most exciting is the flexibility to modify this domain infrastructure. A user interested in looking for different domains can hardwire into the domain class different targets, and create new database tables to be used. Similarly, it would only take a few lines of code within the domain class to tweak the way in which the computer generated domains are formed. This stage of refinement is an area I am excited to explore, but will ultimately not be able to fit within my time as an intern.