A practical primer on processing semantic property norm data

Erin M. Buchanan[1], Simon De Deyne[2], & Maria Montefinese[3]

[1] Harrisburg University of Science and Technology

[2] University of Melbourne

[3] University of Padua

Author Note

Add complete departmental affiliations for each author here. Each new line herein must be indented, like this line.

Enter author note here.

Correspondence concerning this article should be addressed to Erin M. Buchanan, 326 Market St., Harrisburg, PA 17101. E-mail: ebuchanan@harrisburgu.edu

Abstract

Semantic property listing tasks require participants to generate short propositions (e.g., <barks>, <has fur>) for a specific concept (e.g., dog). This task is the cornerstone of the creation of semantic property norms which are essential for modelling, stimuli creation, and understanding similarity between concepts. However, despite the wide applicability of semantic property norms for a large variety of concepts across different groups of people, the methodological aspects of the property listing task have received less attention, even though the procedure and processing of the data can substantially affect the nature and quality of the measures derived from them. The goal of this paper is to provide a practical primer on how to collect and process semantic property norms. We will discuss the key methods to elicit semantic properties and compare different methods to derive meaningful representations from them. This will cover the role of instructions and test context, property pre-processing (e.g., lemmatization), property weighting, and relationship encoding using ontologies. With these choices in mind, we propose and demonstrate a processing pipeline that transparently documents these steps resulting in improved comparability across different studies. The impact of these choices will be demonstrated using intrinsic (e.g. reliability, number of properties) and extrinsic measures (e.g., categorization, semantic similarity, lexical processing). Example data and the impact of choice decisions will be provided. This practical primer will offer potential solutions to several longstanding problems and allow researchers to develop new property listing norms overcoming the constraints of previous studies.

Keywords: semantic, property norm task, tutorial

A practical primer on processing semantic property norm data

1. Available feature norms and their format

- Property listing task original work: (???); (???); (???); (???)
- English: (???), (???), (???), (???), (???)
- Italian: (???); (???), (???)
- German: (???)
- Portuguese: (???)
- Spanish: (???)
- Dutch: (???)
- Blind participants: (???)

I'm sure there are more, here's what we cited recently.

Define concept, feature for clarity throughout - make sure you use these two terms consistently.

2. Pointers about how to collect the data

a. instructions, generation, verification, importance

I really like the way the CSLB did it: https://cslb.psychol.cam.ac.uk/propnorms

They showed the concept, then had a drop down menu for is/has/does, and then the participant typed in a final window. That type of system would solve about half the problems I am going to describe below about using multi-word sequences. Might be some other suggestions, but for that type of processing, you could do combinations and have more consistent data easily.

3. Typical operations performed on features

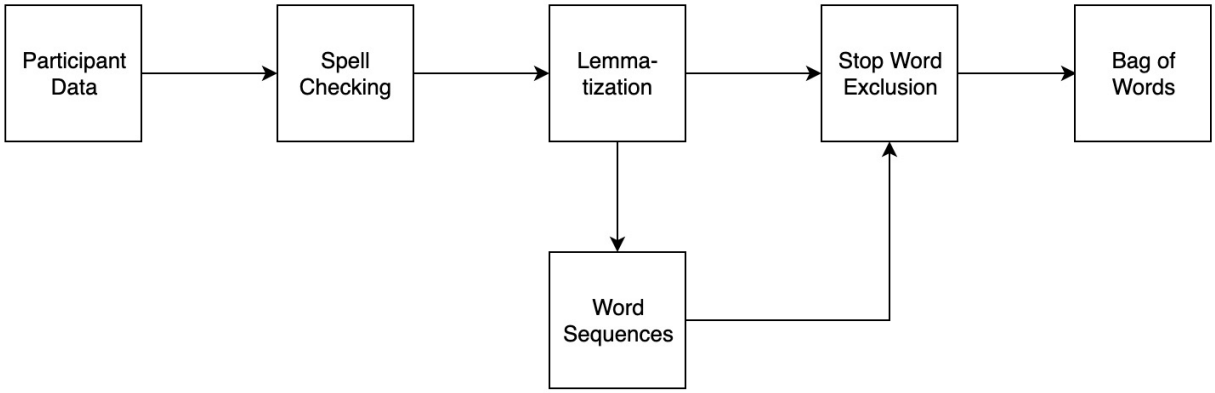

Figure 1. (#fig:flow_chart)Flow chart of proposed semantic processing feature steps.

In the next several sections, we provide a tutorial using *R* on how data from the semantic property norm task might be processed from raw input to finalized output. Figure @ref(fig:flow_chart) portrays the proposed set of steps including spell checking, lemmatization, exclusion of stop words, and final processing in a multi-word sequence approach or a bag of words approach. After detailing these steps, the final data form will compared to previous norms to determine the usefulness of this approach.

Materials and Data Format

The data for this tutorial includes 9553 unique concept-feature responses for 104 concepts from (**???**) that were included in (**???**), (**???**), and (**???**). The data should be structured in tidy format wherein each concept-feature observation is a row and each column is a variable (**???**). Therefore, the data includes a `word` column with the normed concept and an `answer` column with the participant answer.

word	answer
airplane	you fly in it its big it is fast they are expensive they are at an airport you have to be trained to fly it there are lots of seats they get very high up
airplane	wings engine pilot cockpit tail
airplane	wings it flys modern technology has passengers requires a pilot can be dangerous runs on gas used for travel
airplane	wings flys pilot cockpit uses gas faster travel
airplane	wings engines passengers pilot(s) vary in size and color
airplane	wings body flies travel

This data was collected using the instructions provided by (**???**), however, in contrast to the suggestions for consistency detailed above (**???**), each participant was simply given a large text box to include their answer. Each answer includes multiple embedded features, and the tutorial proceeds to demonstrate potential processing addressing the data in this nature. With structured data entry for participants, the suggested processing steps are reduced.

Spelling

Spell checking can be automated with the `hunspell` package in *R* (**???**), which is the spell checking library used in popular programs such as FireFox, Chrome, RStudio, and OpenOffice. Each `answer` can be checked for misspellings across an entire column of answers, which is located in the `master` dataset. The default dictionary is American English, and the `hunspell` vignettes provide details on how to import your own dictionary for non-English languages. The choice of dictionary should also normalize between multiple varieties of the

same language, for example, the `"en_GB"` would convert to British English spellings.

```
## Install the hunspell package if necessary
#install.packages("hunspell")
library(hunspell)
## Check the participant answers
## The output is a list of spelling errors for each line
spelling_errors <- hunspell(master$answer, dict = dictionary("en_US"))
```

The result from the `hunspell()` function is a list object of spelling errors for each row of data. For example, when responding to *apple*, a participant wrote *fruit grocery store orchard red green yelloe good with peanut butter good with caramell*, and the spelling errors were denoted as *yelloe caramell*. After checking for errors, the `hunspell_suggest()` function was used to determine the most likely replacement for each error.

```
## Check for suggestions
spelling_suggest <- lapply(spelling_errors, hunspell_suggest)
```

For *yelloe*, both *yellow yell* were suggested, and *caramel caramels caramel l camellia camel* were suggested for *caramell*. The suggestions are presented in most probable order, and using a few loops with the substitute (`gsub`) function, we can replace all errors with the most likely replacement in a new dataset `spell_checked`. A specialized dictionary with precoded error responses and corrections could be implemented at this stage. Other paid alternatives, such as Bing Spell Check, can be a useful avenue for datasets that may contain brand names (i.e, *apple* versus *Apple*) or slang terms.

```
## Replace with most likely suggestion
spell_checked <- master
### Loop over the data.frame
for (i in 1:nrow(spell_checked)){
  ### See if there are spelling errors
  if (length(spelling_errors[[i]]) > 0) {
```

```
    ### Loop over all errors
    for (q in 1:length(spelling_errors[[i]])){
      ### Replace with the first answer
      spell_checked$answer[i] <- gsub(spelling_errors[[i]][q],
                                      spelling_suggest[[i]][[q]][1],
                                      spell_checked$answer[i])
    }
  }
}
```

Lemmatization

The next step approaches the clustering of word forms into their lemma or head word from a dictionary. The process of lemmatizing words involves using a lexeme set (i.e., all words forms that have the same meaning, *am, are, is*) to convert into a common lemma (i.e., *be*) from a trained dictionary. In contrast, stemming involves processing words using heuristics to remove affixes or inflections, such as *ing* or *s*. The stem or root word may not reflect an actual word in the langauge, as simply removing an affix does not necessarily produce the lemma. For example, in response to *airplane*, *flying* can be easily converted to *fly* by removing the *ing* inflection. However, this same heuristic converts the feature *wings* into *w* after removing both the *s* for a plural marker and the *ing* participle marker. Several packages for *R* include customizable stemmers, notably the `hunspell`, `corpus` (???), and `tm` (???) packages.

Lemmatization is the likely choice for processing property norms, and this process can be achieved by installing `TreeTagger` (???) and the `koRpus` package in *R* (???). TreeTagger is a trained tagger designed to annotate part of speech and lemma information in

text, and parameter files are available for multiple langauges. The koRpus package includes functionality to use TreeTagger in *R*. After installing the package and TreeTagger, we will create a unique set of tokenized words to lemmatize to speed computation.

```
lemmas <- spell_checked
## Install the koRpus package
#install.packages("koRpus")
#install.packages("koRpus.lang.en")
## You must load both packages separately
library(koRpus)
library(koRpus.lang.en)
## Install TreeTagger
#https://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/
## Find all types for faster lookup
all_answers <- tokenize(lemmas$answer, format = "obj", tag = F)
all_answers <- unique(all_answers)
```

The `treetag()` function calls the installation of TreeTagger to provide part of speech tags and lemmas for each token. Importantly, the `path` option should be the directory of the TreeTagger installation.

```
## This function has both suppressWarnings & suppressMessages
## You should first view these to ensure proper processing
temp_tag <- suppressWarnings(
  suppressMessages(
    ## Note: the NULL option is to control for the <unknown> that appears
    ## to occur with the last word in each text
    treetag(c(all_answers, "NULL"),
            ## Control the parameters of treetagger
```

```
            treetagger="manual", format="obj",
            TT.tknz=FALSE, lang="en",
            TT.options=list(path="~/TreeTagger", preset="en"))))
```

This function returns a tagged corpus object, which can be converted into a dataframe of the token-lemma information. The goal would be to replace inflected words with their lemmas, and therefore, unknown values, number tags, and equivalent values are ignored by subseting out these from the dataset.

```
## Remove all tags not using
replacement_lemmas <- temp_tag@TT.res
replacement_lemmas <- subset(replacement_lemmas,
                             #ignore punctuation
                             wclass != "punctuation" &
                             #unknown values
                             lemma != "<unknown>" &
                             #numbers
                             lemma!= "@card@" &
                             #token should change more than case
                             tolower(token) != tolower(lemma))
```

token	tag	lemma	lttr	wclass
is	VBZ	be	2	verb
are	VBP	be	3	verb
trained	VBN	train	7	verb
lots	NNS	lot	4	noun
seats	NNS	seat	5	noun
wings	NNS	wing	5	noun

From this dataset, you can use the `stringi` package (**???**) to replace all of the original tokens with their lemmas. This package allows for replacement lookup across a large set of subsitutions.

```
## Install the stringi package
#install.packages("stringi")
library(stringi)
## Replace all the original tokens with new lemmas using \\b for word boundaries
lemmas$answer <- stri_replace_all_regex(str = lemmas$answer,
                        pattern = paste("\\b", replacement_lemmas$token, "\\b", sep = "")
                        replacement = replacement_lemmas$lemma,
                        vectorize_all = F, list(case_insensitive = TRUE))
```

word	answer
airplane	you fly in it its big it be fast they be expensive they be at an airport you have to be train to fly it there be lot of seat they get very high up
airplane	wing engine pilot cockpit tail
airplane	wing it fly modern technology have passenger require a pilot can be dangerous run on gas use for travel
airplane	wing fly pilot cockpit use gas fast travel
airplane	wing engine passenger pilot(s) vary in size and color
airplane	wing body fly travel

Word Sequences

Multi-word sequences are often coded to mimic a (**???**) style model, with "is-a" and "has-a" type markers. If data were collected to include these markers, this step would be

pre-encoded into the output data, rendering the following code unnecessary. A potential solution for processing messy data could be to search for specific part of speech sequences that mimic the "is-a" and "has-a" strings. An examination of the coding in (???) and (???) indicates that the feature tags are often verb-noun or verb-adjective-noun sequences. Using TreeTagger on each concept's answer set, we can obtain the parts of speech in context for each lemma. With `dplyr` (???), new columns are added to tagged data to show all bigram and trigram sequences. All verb-noun and verb-adjective-noun combinations are selected, and any words not part of these multi-word sequences are treated as unigrams. Finally, the `table()` function is used to tabulate the final count of n-grams and their frequency.

```
multi_words <- data.frame(Word=character(),
                          Feature=character(),
                          Frequency=numeric(),
                          stringsAsFactors=FALSE)
unique_concepts <- unique(lemmas$word)
## Install dplyr
#install.packages("dplyr")
library(dplyr)
## Loop over each word
for (i in 1:length(unique_concepts)){
  ## Create parts of speech for clustering together
  temp_tag <- suppressWarnings(
    suppressMessages(
      treetag(c(lemmas$answer[lemmas$word  == unique_concepts[i]], "NULL"),
          ## Control the parameters of treetagger
          treetagger="manual", format="obj",
          TT.tknz=FALSE, lang="en",
          TT.options=list(path="~/TreeTagger", preset="en"))))
```

```
## Save only the data.frame, remove NULL
temp_tag <- temp_tag@TT.res[-nrow(temp_tag@TT.res) , ]
## Subset out information you don't need
temp_tag <- subset(temp_tag,
                   wclass != "comma" & wclass != "determiner" &
                     wclass != "preposition" & wclass != "modal" &
                     wclass != "predeterminer" & wclass != "particle" &
                     wclass != "to" & wclass != "punctuation" &
                     wclass != "fullstop" & wclass != "conjunction" &
                     wclass != "pronoun")
## Create a temporary tibble
temp_tag_tibble <- as_tibble(temp_tag)
## Create part of speech and features combined
temp_tag_tibble <- mutate(temp_tag_tibble,
                          two_words = paste(token,
                                            lead(token), sep = "_"))
temp_tag_tibble <- mutate(temp_tag_tibble,
                          three_words = paste(token,
                                              lead(token), lead(token, n = 2L), sep =
temp_tag_tibble <- mutate(temp_tag_tibble,
                          two_words_pos = paste(wclass,
                                                lead(wclass), sep = "_"))
temp_tag_tibble <- mutate(temp_tag_tibble,
                          three_words_pos = paste(wclass,
                                                  lead(wclass), lead(wclass, n = 2L),
## Find verb noun or verb adjective nouns to cluster on
verb_nouns <- grep("\\bverb_noun", temp_tag_tibble$two_words_pos)
```

```
  verb_adj_nouns <- grep("\\bverb_adjective_noun", temp_tag_tibble$three_words_pos)
  ## Use combined and left over features
  features_for_table <- c(temp_tag_tibble$two_words[verb_nouns],
                          temp_tag_tibble$three_words[verb_adj_nouns],
                          temp_tag_tibble$token[-c(verb_nouns, verb_nouns+1,
                                                   verb_adj_nouns, verb_adj_nouns+1,
                                                   verb_adj_nouns+2)])
  ## Create a table of frequencies
  word_table <- as.data.frame(table(features_for_table))
  ## Clean up the table
  word_table$Word <- unique_concepts[i]
  colnames(word_table) = c("Feature", "Frequency", "Word")
  multi_words <- rbind(multi_words, word_table[ , c(3, 1, 2)])
}
```

This procedure does produce some positive output, such as *fingers-have_fingernails* and *couches-have_cushions*. One obvious limitation is the potential necessity to match this coding system to previous codes, which were predominately hand processed. Further, many similar phrases, such as the ones for *zebra* shown below may require fuzzy logic matching to ensure that the different codings for *is-a-horse* are all combined together.

Word	Feature	Frequency
zebra	2	1
zebra	4	6
zebra	4-leg	1
zebra	ace	1
zebra	Africa	28
zebra	animal	43
zebra	area	1
zebra	around	1
zebra	baby	1
zebra	be	21
zebra	be_animal	8
zebra	be_animal_life	1
zebra	be_black_white	2
zebra	be_exotic_leg	1
zebra	be_funny_character	1
zebra	be_grass	1
zebra	be_herbivore	2
zebra	be_horse	1
zebra	be_inhabitant	1
zebra	be_logo	2
zebra	be_mama	1
zebra	be_mammal	4
zebra	be_omnivore	1
zebra	be_plastic_kitchen	1
zebra	be_plastic_metal	1
zebra	be_similar_horse	1
zebra	be_stripe	2

```
library(stopwords)

## Remove stop words

multi_words <- subset(multi_words,

                      !(Feature %in% stopwords(language = "en", source = "snowball")))
```

b. Weighting

c. feature type ontologies

d. identify cut off for idiosyncratic features (should it be necessary?)

4. Specification of how this is automated (package description)

a. tests to see if things work: e.g. manual spell checks vs automated ones

5. Evaluation of the approach

a. internal (quality, size, consistency) - ?

b. feature size number of features work

ii. classifier for ontology, compare results to previous work

b. externally (categorization, similarity) – MEN dataset, Lapata categorization task

6. Challenges and opportunities

Discussion

References