Quick Start Guide

Preliminaries

conText was designed to work closely with quanteda, as such, most functions will expect a quanteda object such as a *tokens* or *dfm* or *fcm* object. To make the most of **conText**, we recommend that you familiarize yourself with some of quanteda's basic functionalities (see Quanteda Quick Start Guide).

Setup

Installing the package

You can install the package directly from CRAN:

```
install.packages("conText")
```

Or, for the latest development version, from GitHub:

```
devtools::install_github("prodriguezsosa/conText")
```

Load package

```
library(conText)

# other libraries used in this guide
library(quanteda)
library(dplyr)
library(text2vec)
```

Data

To use **conText** you will need three objects:

1. A (quanteda) **corpus** with the documents and corresponding document variables (covariates) you want to evaluate.
2. A set of (GloVe) **pre-trained embeddings**.
3. A **transformation matrix** specific to the pre-trained embeddings.

In this guide we will use the sample objects included in the package: `cr_sample_corpus`, `cr_glove_subset` and `cr_transform`. Note, these are only meant to illustrate function implementations, keep in mind their (small) size when interepreting results. We've made available the full versions of these objects here.

Pre-processing

Pre-processing is always context-specific (no pun intended). We generally recommend not "overly" pre-processing a corpus. Below we outline how a typical pre-processing pipeline might look. Note, `cr_sample_corpus` already has some minor pre-processing so some of these steps may be redundant but are left for completeness. It is generally a good idea to set `padding = TRUE` in `tokens_select()` to avoid

making non-adjacent words adjacent prior to computing the embeddings. This is increasingly important as more features are removed.

```
# tokenize corpus removing unnecessary (i.e. semantically uninformative) elements
toks <- tokens(cr_sample_corpus, remove_punct=T, remove_symbols=T, remove_numbers=T, remove_separators='

# clean out stopwords and words with 2 or fewer characters
toks_nostop <- tokens_select(toks, pattern = stopwords("en"), selection = "remove", min_nchar=3)

# only use features that appear at least 5 times in the corpus
feats <- dfm(toks_nostop, tolower=T, verbose = FALSE) %>% dfm_trim(min_termfreq = 5) %>% featnames()

# leave the pads so that non-adjacent words will not become adjacent
toks <- tokens_select(toks_nostop, feats, padding = TRUE)
```

The building blocks of 'a la carte' embeddings

Suppose we are interested in the semantics surrounding the word "immigration" in the U.S. Congress during the Obama years (sessions 111th - 114th).

1. Build a (tokenized) corpus of contexts

We begin by identifying all instances of the *target* term –"immigration"– in our corpus and, for each instance, store it's context– the N words (N = 6 in this example) preceding and following the instance, and tokenize. Given our tokenized corpus, `toks`, we can do this in one step using `conText::tokens_context()` –a wrapper function for quanteda::kwic(). Notice, both the input, `x`, and the output are a quanteda `tokens` objects. Each document in `immig_toks` –tokens of a context around an instance of *immigration*– inherits the document variables (`docvars`) of the document from whence it came, along with a column registering the corresponding pattern. This information can be retrieved using `docvars()`.

```
# build a tokenized corpus of contexts sorrounding the target term "immigration"
immig_toks <- tokens_context(x = toks, pattern = "immigr*", window = 6L)
```

```
## 125 instances of "immigrant" found.
## 288 instances of "immigrants" found.
## 924 instances of "immigration" found.
```

```
head(docvars(immig_toks), 3)
```

```
##       pattern party gender nominate_dim1
## 1  immigrants     D      F        -0.759
## 2 immigration     D      F        -0.402
## 3   immigrant     D      F        -0.402
```

2. Build a document-feature-matrix

Given a tokenized corpus of contexts, we next build it's corresponding document-feature-matrix, namely a matrix where each row represents a given document's vector of feature counts. We do this using quanteda's `dfm()` function.

```
# build document-feature matrix
immig_dfm <- dfm(immig_toks)
immig_dfm[1:3,1:3]
```

```
## Document-feature matrix of: 3 documents, 3 features (66.67% sparse) and 4 docvars.
##        features
```

```
## docs    institutions moral stood
##   text1            1     1     1
##   text2            0     0     0
##   text3            0     0     0
```

3. Build a document-embedding-matrix

Given a 'dfm', `immig_dfm`, a set of pre-trained embeddings, `cr_glove_subset` and a corresponding transformation matrix, `cr_transform`, we can proceed to embed each document –i.e. context– '*a la carte*'. We embed a document by multiplying each of it's feature counts with their corresponding pre-trained feature-embeddings, column-averaging the resulting vectors, and multiplying by the transformation matrix. This "transforms" a *sparse* V-dimensional vector (a vector of feature counts, with V = number of features in the corpus) into a *dense* D-dimensional vector (a D-dimensional embedding, with D = dimensions of the pre-trained embeddings). We do this using `conText::dem()`–'dem' standing for *document-embedding-matrix*. Each row in this matrix represents an 'a la carte' (ALC) embedding of a single instance of "immigration".

Keep in mind, only those features that appear in the set of pre-trained embeddings will be used in computing a document's embedding. Documents with no features overlapping with the pre-trained provided are dropped. `dem()` –outputs a 'dem-class' object which is similar in many ways to 'dfm-class' object. Importantly, a 'dem-class' object inherits all the documenta variables, 'docvars', from the 'dfm' used to compute it (except those of documents that could not be embedded). Additionally, a 'dem-class' object will store other useful attributes, including the names of the documents that were embedded and the vector of features used.

```
# build a document-embedding-matrix
immig_dem <- dem(x = immig_dfm, pre_trained = cr_glove_subset, transform = TRUE, transform_matrix = cr_

# each document inherits its corresponding docvars
#head(immig_dem@docvars)

# you can check which documents were not embedded due to lack of overlapping features (in this example
# note: 'quanteda' functions like `docvars()` and `docnames()` don't work on `dem` objects, so you will
#setdiff(docnames(immig_dfm), immig_dem@Dimnames$docs)

# vector of features used to create the embeddings
#head(immig_dem@features)
```

4. Average over document embeddings

We now have an ALC embedding for each instance of "immigration" in our sample corpus. To get a single corpus-wide ALC embedding for "immigration", we can simply take the column-average of the single-instance ALC embeddings.

```
## [1]   1 300
```

However, we are usually interested in exploring semantic differences across groups. To do so, we can average using a grouping variable defined by one or a combination of the 'docvars'. We do this using `conText::dem_group()` (very similar in flavor to quanteda's `dfm_group()`). In our example below, this results in an ALC embedding of "immigration" for each party, hence the dimensions 2 by 300.

```
# to get group-specific embeddings, average within party
immig_wv_party <- dem_group(immig_dem, groups = immig_dem@docvars$party)
dim(immig_wv_party)
```

```
## [1]   2 300
```

5. Comparing group embeddings

Given an ALC embedding for each party, we can proceed to do some exploratory analysis.

Nearest neighbors

First, we can evaluate differences in nearest neighbors –features with the highest cosine-similarity with each group embedding– using `conText::nns()`. We use the `candidates` argument to limit the set of features we want `nns` to consider as candidate nearest neighbors. In this example we limit candidates to those features that appear in our corpus (otherwise, any word in the set of pre-trained embeddings can be a candidate).

```
# find nearest neighbors by party
# setting as_list = FALSE combines each group's results into a single tibble (useful for joint plotting
immig_nns <- nns(immig_wv_party, pre_trained = cr_glove_subset, N = 5, candidates = immig_wv_party@featu

# check out results for Republican party
immig_nns[["R"]]
```

```
## # A tibble: 5 x 4
##   target feature      rank value
##   <chr>  <chr>       <int> <dbl>
## 1 R      immigration     1 0.843
## 2 R      illegal         2 0.768
## 3 R      immigrants      3 0.732
## 4 R      illegally       4 0.664
## 5 R      amnesty         5 0.658
```

Cosine similarity

We can also look at the relationship between each group's ALC embedding of "immigration" and a specific set of features. For example, how does each party's understanding of immigration compare on the "reform" vs. "enforcement" dimension? To answer this, we use `conText::cos_sim()`. Results (on this limited sample of the data) suggest Democrats are more prone to speak of "reform" in the context of "immigration" whereas Republicans are more likely to speak of "enforcement".

```
# compute the cosine similarity between each party's embedding and a specific set of features
cos_sim(immig_wv_party, pre_trained = cr_glove_subset, features = c('reform', 'enforcement'), as_list =
```

```
##   target     feature     value
## 1      D      reform 0.5949757
## 2      R      reform 0.4270958
## 3      D enforcement 0.6059709
## 4      R enforcement 0.5968879
```

Nearest neighbors cosine similarity ratio

A third exploratory function included in 'conText' is `nns_ratio()`. Given ALC embeddings for two groups, `nns_ratio()` computes the ratio of cosine similarities between group embeddings and features –that is, for any given feature it first computes the similarity between that feature and each group embedding, and then takes the ratio of these two similarities. This ratio captures how "discriminant" a feature is of a given group. Values larger (smaller) than 1 mean the feature is more (less) discriminant of the group in the numerator (denominator). Use the `numerator` argument to define which group represents the numerator in this ratio. If `N` is defined, this ratio is computed for union of the top `N` nearest neighbors.

```
# compute the cosine similarity between each party's embedding and a specific set of features
nns_ratio(x = immig_wv_party, N = 10, numerator = "R", candidates = immig_wv_party@features, pre_traine
```

```
##          feature     value
```

```
## 1        enforce 1.2076754
## 2        illegal 1.1834716
## 3        amnesty 1.1716428
## 4      illegally 1.1232888
## 5           laws 1.0893494
## 6          legal 1.0261909
## 7     immigrants 1.0070933
## 8         border 1.0043546
## 9    enforcement 0.9850108
## 10   immigration 0.9809758
## 11     immigrant 0.9118817
## 12  undocumented 0.9008638
## 13        broken 0.7631982
## 14 comprehensive 0.7248641
## 15        reform 0.7178373
```

Nearest contexts

We are used to hearing about "nearet neighbors" when it comes to interpreting word embeddings. However, it is often the case that nearest neighbors –i.e. single words– without any context are hard to interpret. For this reason we introduce "nearest contexts" – contexts around a target term that are semantically close –i.e. high cosine similarity– to the ALC embedding of that term. To explore nearest contexts we use the `ncs()` function. `ncs()` computes the cosine similarities between the ALC (group) embeddings and the single-instance ALC embeddings of the underlying contexts.

```
# compute the cosine similarity between each party's embedding and a set of tokenized contexts
immig_ncs <- ncs(x = immig_wv_party, contexts_dem = immig_dem, contexts = immig_toks, N = 5, as_list = T

# nearest contexts to Republican embedding of target term
immig_ncs[["R"]]
```

```
## # A tibble: 5 x 4
##   target context                                                  rank value
##   <chr>  <chr>                                                   <int> <dbl>
## 1 R      america suggest immigration can good thing illegal immigra~     1 0.817
## 2 R      immigration can good thing immigration illegal legal immig~     2 0.787
## 3 R      immigration law prior illegal immigration reform responsib~     3 0.785
## 4 R      good thing immigration illegal immigration legal mean mean~     4 0.773
## 5 R      going cost hardworking taxpayers america suggest can good ~     5 0.758
```

Stemming

All functions to explore nearest neighbors –`nns`, `cos_sim`, `nns_ratio`– have to option to add stemming. This can be useful to group nearest neighbors with the same stem e.g. "reform", "reforms", "reformed". Under the hood, candidate nearest neighbors are stemmed using the `Snowball` library and cosine similarities with the target embeddings are averaged across nearest neighbors with the same stem. To avoid noisy words influencing this average e.g. "reformedthesystem", we recommend you remove misspelled words from the candidate set `candidates` (you can automate this using the `hunspell` library). See example below.

```
# consider removing noisy candidate nearest neighbors
library(hunspell) # spellcheck library
```

```
##
## Attaching package: 'hunspell'
```

```
## The following object is masked from 'package:quanteda':
##
```

```
##     dictionary
```

```
# extract candidate features from the dem object
immig_feats <- immig_wv_party@features

# check spelling. toupper avoids names being considered misspelled
spellcheck <-  hunspell_check(toupper(immig_feats), dict = hunspell::dictionary("en_US")) #
immig_feats <- immig_feats[spellcheck]

# find nearest neighbors by party using stemming
immig_nns_stem <- nns(immig_wv_party, pre_trained = cr_glove_subset, N = 5, candidates = immig_feats, s

# check out results for Republican party
immig_nns_stem[["R"]]
```

```
## # A tibble: 5 x 4
##   target feature  rank value
##   <chr>  <chr>   <int> <dbl>
## 1 R      illeg       1 0.716
## 2 R      immigr      2 0.704
## 3 R      amnesti     3 0.658
## 4 R      enforc      4 0.593
## 5 R      law         5 0.578
```

Multiple Keywords

In the above example we had one target word, "immigration". However, we can also explore the semantics of multiple targets simultaneously, including phrases! We simply provide a vector of patterns in the `pattern` argument of `tokens_context()`.

```
# build a corpus of contexts sorrounding the target term "immigration"
mkws_toks <- tokens_context(x = toks, pattern = c("immigration", "welfare", "immigration reform", "econ

# create document-feature matrix
mkws_dfm <- dfm(mkws_toks)

# create document-embedding matrix using a la carte
mkws_dem <- dem(x = mkws_dfm, pre_trained = cr_glove_subset, transform = TRUE, transform_matrix = cr_tra

# get embeddings for each pattern
mkws_wvs <- dem_group(mkws_dem, groups = mkws_dem@docvars$pattern)

# find nearest neighbors for each keyword
mkws_nns <- nns(mkws_wvs, pre_trained = cr_glove_subset, N = 5, candidates = mkws_wvs@features, as_list

# to check results for a given pattern
mkws_nns[["immigration reform"]]
```

```
## # A tibble: 5 x 4
##   target             feature        rank value
##   <chr>              <chr>         <int> <dbl>
## 1 immigration reform comprehensive     1 0.817
## 2 immigration reform immigration       2 0.739
## 3 immigration reform reform            3 0.635
## 4 immigration reform pass              4 0.609
## 5 immigration reform bipartisan        5 0.581
```

This can also be useful if we wanted to explore *group differences* in the semantics of a given *topic* where we define a *topic* by a vector of *topical* target words. Note, these *topical* embeddings are computed off of the collection of contexts around the set of pattern words provided. So, in the example below, the contexts around each of "immigration", "immigrant" and "immigration reform" are treated as a single collection of contexts for the purposes of computing each party's *topical* embedding.

```
# build a corpus of contexts sorrounding the immigration related words
topical_toks <- tokens_context(x = toks, pattern = c("immigration", "immigrant", "immigration reform"),

# create document-feature matrix
topical_dfm <- dfm(topical_toks)

# create document-embedding matrix using a la carte
topical_dem <- dem(x = topical_dfm, pre_trained = cr_glove_subset, transform = TRUE, transform_matrix =

# get "topical" embeddings for each party
topical_wvs <- dem_group(topical_dem, groups = topical_dem@docvars$party)

# find nearest neighbors for each keyword
nns(topical_wvs, pre_trained = cr_glove_subset, N = 5, candidates = topical_wvs@features, stem = TRUE, a
```

```
## # A tibble: 10 x 4
##    target feature     rank value
##    <fct>  <chr>      <int> <dbl>
##  1 D      comprehens     1 0.699
##  2 R      illeg          1 0.677
##  3 R      immigr         2 0.673
##  4 D      broken         2 0.668
##  5 D      immigr         3 0.663
##  6 D      reform         4 0.661
##  7 R      amnesti        3 0.659
##  8 R      enforc         4 0.626
##  9 R      law            5 0.609
## 10 D      act            5 0.549
```

Wrapper functions

The above functions give users a lot of flexibility, however, it can be cumbersome to write each step every time one wants to do some exploratory analysis on a corpus. In this section we'll look into a collection of wrapper functions that allow users to go straight from a tokenized corpus to results. Each of the four exploratory analysis functions –`nns()`, `cos_sim()`, `nns_ratio()` and `ncs()`– described above has its corresponding wrapper function.

An key advantage of the wrapper functions is that they make it easy to obtain standard errors around the sample statistics of interest via *bootstrapping*. Bootstrapping works by sampling with replacement documents from our tokenized corpus of contexts (within group if the argument `groups` is specified) and going through the above steps for each of our exploratory analysis functions (i.e. computing an 'a la carte' embedding on each bootstrapped sample, then computing the cosine similarities etc.). Let's start with `conText::get_nns()`, a wrapper function for `nns()`.

Nearest neighbors

We use the same tokenized corpus of "immigration" contexts –`immig_toks`– as above, and again limit candidate nearest neighbors to the set of features in our corpus. To group by *party* we set `groups = docvars(immig_corpus, 'party')` –the `groups` argument can include more than two groups. If no grouping

variable is provided, the full set of (tokenized) contexts are aggregated into one single embedding. To estimate standard errors for the cosine similarities between each party's embedding and their corresponding nearest neighbors we set `bootstrap = TRUE` and define the desired number of bootstraps with `num_bootstrap`. Notice the output now has an additional column `std.error`. This is simply the standard deviation of the sampling distribution of cosine similarities obtained via bootstrapping. Note, values may differ slightly to the step-by-step process outlined above as they represent averages over bootstrapped samples.

```
# we limit candidates to features in our corpus
feats <- featnames(dfm(immig_toks))

# compare nearest neighbors between groups
set.seed(2021L)
immig_party_nns <- get_nns(x = immig_toks, N = 10,
        groups = docvars(immig_toks, 'party'),
        candidates = feats,
        pre_trained = cr_glove_subset,
        transform = TRUE,
        transform_matrix = cr_transform,
        bootstrap = TRUE,
        num_bootstraps = 10,
        as_list = TRUE)

# nearest neighbors of "immigration" for Republican party
immig_party_nns[["R"]]
```

```
## # A tibble: 10 x 5
##    target feature       rank value std.error
##    <chr>  <chr>        <int> <dbl>     <dbl>
##  1 R      immigration      1 0.842   0.00565
##  2 R      illegal          2 0.758   0.0108
##  3 R      immigrants       3 0.739   0.00899
##  4 R      illegally        4 0.661   0.0106
##  5 R      amnesty          5 0.655   0.0128
##  6 R      laws             6 0.589   0.0132
##  7 R      enforcement      7 0.588   0.0116
##  8 R      enforce          8 0.577   0.0124
##  9 R      legal            9 0.567   0.0125
## 10 R      undocumented    10 0.565   0.00984
```

Cosine similarity

`conText::get_cos_sim()` is a wrapper function for `cos_sim()`, used to evaluate how similar each group's (or single if `groups` is not defined) embedding is to a set of features of interest –as with `get_nns()`, the `groups` argument can take on more than two groups. Again we set `bootstrap = TRUE` to obtain standard errors for the cosine similarities.

```
# compute the cosine similarity between each group's embedding and a specific set of features
set.seed(2021L)
get_cos_sim(x = immig_toks,
            groups = docvars(immig_toks, 'party'),
            features = c("reform", "enforce"),
            pre_trained = cr_glove_subset,
            transform = TRUE,
            transform_matrix = cr_transform,
            bootstrap = TRUE,
            num_bootstraps = 10,
```

```
            as_list = FALSE)
```

```
## # A tibble: 4 x 4
##   target feature value std.error
##   <fct>  <fct>   <dbl>     <dbl>
## 1 D      reform  0.592   0.0191
## 2 D      enforce 0.484   0.00976
## 3 R      reform  0.427   0.0159
## 4 R      enforce 0.577   0.0124
```

Nearest neighbors cosine similarity ratio

`conText::get_nns_ratio()` is a wrapper function for `nns_ratio()`, used to gauge how discriminant nearest neighbors are of each group. Unlike `get_nns()` and `get_cos_sim()`, a `groups` argument must be provided and it must be binary. As with `nns_ratio()`, we use the `numerator` argument to control which group to use as the numerator in the underlying ratio. We again limit the candidate nearest neighbors to the set of local features and set `bootstrap = TRUE` to get standard errors, in this case of the cosine similarity ratios.

Finally, `get_nns_ratio()` allows us to make inferences using a permutation test, specifically around the *absolute deviation of the observed cosine similarity ratio from* 1 (this captures how discriminant a given nearest neighbor is). Specifically, for each permutation, the grouping variable is randomly shuffled and the absolute deviation of the cosine similarity ratios from 1 is computed. The empirical p.value is then the proportion of these "permuted" deviations that are larger than the observed deviation.

The output of `get_nns_ratio()` contains three additional columns (relative to `nns_ratio()`) if `bootstrap = TRUE` and `permute = TRUE`. These are the standard errors around the cosine similarity ratios, the corresponding (empirical) `p.value` and a `group` variable identifying which group the nearest neighbor belonged to – `shared` means it appeared in both groups' top N nearest neighbors.

```
# we limit candidates to features in our corpus
feats <- featnames(dfm(immig_toks))

# compute ratio
set.seed(2021L)
immig_nns_ratio <- get_nns_ratio(x = immig_toks,
              N = 10,
              groups = docvars(immig_toks, 'party'),
              numerator = "R",
              candidates = feats,
              pre_trained = cr_glove_subset,
              transform = TRUE,
              transform_matrix = cr_transform,
              bootstrap = TRUE,
              num_bootstraps = 10,
              permute = TRUE,
              num_permutations = 10,
              verbose = FALSE)
```

```
## starting bootstraps
## done with bootstraps
## starting permutations
## done with permutations
```

```
head(immig_nns_ratio)
```

```
## # A tibble: 6 x 5
##   feature   value std.error p.value group
```

```
##   <chr>     <dbl>     <dbl>   <dbl> <chr>
## 1 enforce    1.19    0.0306     0   R
## 2 illegal    1.17    0.0233     0   shared
## 3 amnesty    1.17    0.0211     0   R
## 4 illegally  1.13    0.0246     0   shared
## 5 laws       1.08    0.0424     0   R
## 6 legal      1.03    0.0382     0.1 R
```

conText also includes a plotting function, `plot_nns_ratio`, specific to `get_nns_ratio()`, providing a nice visualization of its output. `alpha` defines the desired significance threshold to denote "significant" results on the plot (indicated by a $*$ next to the feature). Also, you can choose between two different visualizations of the same results using the `horizontal` argument.

```
plot_nns_ratio(x = immig_nns_ratio, alpha = 0.01, horizontal = TRUE)
```

Nearest contexts

`conText::get_ncs()` is a wrapper function for `ncs()`, used to compute cosine similarities between ALC (group) embeddings and the ALC embeddings of individual contexts with the option to bootstrap standard errors.

```
# compare nearest neighbors between groups
set.seed(2021L)
immig_party_ncs <- get_ncs(x = immig_toks,
                           N = 10,
                           groups = docvars(immig_toks, 'party'),
                           pre_trained = cr_glove_subset,
                           transform = TRUE,
                           transform_matrix = cr_transform,
                           bootstrap = TRUE,
                           num_bootstraps = 10,
                           as_list = TRUE)

# nearest neighbors of "immigration" for Republican party
immig_party_ncs[["R"]]
```

```
## # A tibble: 10 x 5
##    target context                                          rank value std.error
##    <chr>  <chr>                                           <int> <dbl>     <dbl>
##  1 R      america suggest immigration can good thing ille~     1 0.815   0.00830
##  2 R      immigration can good thing immigration illegal ~     2 0.789   0.00880
##  3 R      immigration law prior illegal immigration refor~     3 0.783   0.00917
##  4 R      good thing immigration illegal immigration lega~     4 0.771   0.00717
##  5 R      going cost hardworking taxpayers america sugges~     5 0.756   0.00921
##  6 R      actually increase illegal immigration reducing ~     6 0.734   0.00561
##  7 R      immigration enforcement along southwest border ~     7 0.728   0.00697
##  8 R      late right thing hold president responsible pol~     8 0.721   0.00785
##  9 R      backlog met likely lead reduction legal increas~     9 0.719   0.0111
## 10 R      responsible immigration policies ignoring immig~    10 0.717   0.00942
```

Embedding regression

The above framework allows us to explore semantic differences over one grouping variable at a time. However, we often want to look at covariate effects while controlling for other covariates or indeed go beyond discrete covariates. In Rodriguez, Spirling and Stewart (2021) we show how `a la carte` embeddings can be used

within a regression-framework that allows us to do just that. The corresponding package function is `conText::conText()`.

`conText()` tries to follow a similar syntax as R's `lm()` and `glm()` functions. `data` must be a quanteda tokens object with covariates stored as document variables (`docvars`). We next specify a formula consisting of the target word of interest, e.g. "immigration" and the set of covariates. To use all covariates in `data`, we can speficy `immigration ~ .`. `formula` can also take vectors of target words e.g. `c("immigration", "immigrants") ~ party + gender` and phrases e.g. `"immigration reform" ~ party + gender` – place phrases in quotation marks.

```
# two factor covariates
set.seed(2021L)
model1 <- conText(formula = immigration ~ party + gender,
                  data = toks,
                  pre_trained = cr_glove_subset,
                  transform = TRUE, transform_matrix = cr_transform,
                  bootstrap = TRUE, num_bootstraps = 10,
                  permute = TRUE, num_permutations = 100,
                  window = 6, case_insensitive = TRUE,
                  verbose = FALSE)
```

```
##   coefficient normed.estimate  std.error p.value
## 1     party_R       0.5988860 0.05041898       0
## 2    gender_M       0.5355259 0.03126510       0
```

```
# notice, non-binary covariates are automatically "dummified"
rownames(model1)
```

```
## [1] "party_R"     "gender_M"    "(Intercept)"
```

`conText()` outputs a `conText-class` object which is simply a `dgCMatrix class` matrix corresponding to the beta coefficients (ALC embeddings) with additional attributes including: a table (automatically printed) with the normed coefficients (excluding the intercept), their std. errors and p-values, `@normed_coefficients`, and the set of features used when creating the embeddings `@features`.

We can combine the coefficients to compute the ALC embeddings for the various combinations of the covariates (see Rodriguez, Spirling and Stewart (2021) for details).

```
# D-dimensional beta coefficients
# the intercept in this case is the ALC embedding for female Democrats
# beta coefficients can be combined to get each group's ALC embedding
DF_wv <- model1['(Intercept)',] # (D)emocrat - (F)emale
DM_wv <- model1['(Intercept)',] + model1['gender_M',] # (D)emocrat - (M)ale
RF_wv <- model1['(Intercept)',] + model1['party_R',]  # (R)epublican - (F)emale
RM_wv <- model1['(Intercept)',] + model1['party_R',] + model1['gender_M',] # (R)epublican - (M)ale

# nearest neighbors
nns(rbind(DF_wv,DM_wv), N = 10, pre_trained = cr_glove_subset, candidates = model1@features)
```

```
## $DM_wv
## # A tibble: 10 x 4
##    target feature        rank value
##    <chr>  <chr>         <int> <dbl>
##  1 DM_wv  immigration       1 0.844
##  2 DM_wv  broken            2 0.693
##  3 DM_wv  reform            3 0.660
##  4 DM_wv  comprehensive     4 0.646
##  5 DM_wv  immigrants        5 0.596
```

```
##  6 DM_wv  illegal           6 0.589
##  7 DM_wv  enforcement       7 0.567
##  8 DM_wv  border            8 0.555
##  9 DM_wv  system            9 0.545
## 10 DM_wv  fix              10 0.538
##
## $DF_wv
## # A tibble: 10 x 4
##    target feature        rank value
##    <chr>  <chr>         <int> <dbl>
##  1 DF_wv  immigration       1 0.835
##  2 DF_wv  comprehensive     2 0.641
##  3 DF_wv  enforcement       3 0.640
##  4 DF_wv  immigrants        4 0.629
##  5 DF_wv  broken            5 0.613
##  6 DF_wv  reform            6 0.608
##  7 DF_wv  law               7 0.595
##  8 DF_wv  illegal           8 0.589
##  9 DF_wv  laws              9 0.567
## 10 DF_wv  border           10 0.565
```

To access the normed coefficients for plotting:

```
model1@normed_cofficients
```

```
##   coefficient normed.estimate  std.error p.value
## 1     party_R       0.5988860 0.05041898       0
## 2    gender_M       0.5355259 0.03126510       0
```

conText can also take continuous covariates. In the example below we estimate a model using the first dimension of the NOMINATE score–understood to capture the Liberal-Conservative spectrum on economic matters. To explore similarity with specific features we use the fitted model to compute the fitted values –ALC embeddings– at various percentiles of the NOMINATE score. Consistent with our results above, the higher the NOMINATE score (more Conservative) the greater the similarity between the ALC embedding for `immigration` and the feature `enforcement` whereas the lower the NOMINATE score (more Liberal) the greater its similarity with the feature `reform`.

```
# continuous covariate
set.seed(2021L)
model2 <- conText(formula = immigration ~ nominate_dim1,
                  data = toks,
                  pre_trained = cr_glove_subset,
                  transform = TRUE, transform_matrix = cr_transform,
                  bootstrap = TRUE, num_bootstraps = 10,
                  permute = TRUE, num_permutations = 100,
                  window = 6, case_insensitive = TRUE,
                  verbose = FALSE)
```

```
##     coefficient normed.estimate  std.error p.value
## 1 nominate_dim1       0.6720702 0.06397647       0
```

```
# look at percentiles of nominate
percentiles <- quantile(docvars(cr_sample_corpus)$nominate_dim1, probs = seq(0.05,0.95,0.05))
percentile_wvs <- lapply(percentiles, function(i) model2["(Intercept)",] + i*model2["nominate_dim1",]) %
percentile_sim <- cos_sim(x = percentile_wvs, pre_trained = cr_glove_subset, features = c("reform", "en

# check output
```

```r
rbind(head(percentile_sim[["reform"]], 5),tail(percentile_sim[["reform"]], 5))
```

```
##    target feature     value
## 1      5%  reform 0.6645026
## 2     10%  reform 0.6598026
## 3     15%  reform 0.6557875
## 4     20%  reform 0.6532968
## 5     25%  reform 0.6501394
## 15    75%  reform 0.5045321
## 16    80%  reform 0.4909143
## 17    85%  reform 0.4841337
## 18    90%  reform 0.4787336
## 19    95%  reform 0.4735839
```

```r
rbind(head(percentile_sim[["enforce"]], 5),tail(percentile_sim[["enforce"]], 5))
```

```
##    target feature     value
## 20     5% enforce 0.4558998
## 21    10% enforce 0.4695425
## 22    15% enforce 0.4797996
## 23    20% enforce 0.4856559
## 24    25% enforce 0.4926141
## 34    75% enforce 0.6294170
## 35    80% enforce 0.6346377
## 36    85% enforce 0.6369822
## 37    90% enforce 0.6387348
## 38    95% enforce 0.6403148
```

Local GloVe and transformation matrix

If (a) you have a large enough corpus to train a full GloVe embeddings model and (b) your corpus is distinctive in some way –e.g. a collection of articles from scientific journals–, then you may want to consider estimating your own set of embeddings and transformation matrix –otherwise you should be good to go using GloVe pre-trained embeddings. The first step is to estimate GloVe embeddings on the full corpus which you will then use as your pre-trained embeddings.

Estimate GloVe embeddings

This example is taken (with minor changes) from this quanteda vignette on computing GloVe embeddings using `quanteda` and `text2vec`.

```r
library(text2vec)

#----------------------------------
# estimate glove model
#----------------------------------

# construct the feature co-occurrence matrix for our toks object (see above)
toks_fcm <- fcm(toks, context = "window", window = 6, count = "frequency", tri = FALSE) # important to

# estimate glove model using text2vec
glove <- GlobalVectors$new(rank = 300,
                           x_max = 10,
                           learning_rate = 0.05)
```

```
wv_main <- glove$fit_transform(toks_fcm, n_iter = 10,
                               convergence_tol = 1e-3,
                               n_threads = 2) # set to 'parallel::detectCores()' to use all available c
```

```
## INFO  [16:03:11.244] epoch 1, loss 0.2267
## INFO  [16:03:13.503] epoch 2, loss 0.0768
## INFO  [16:03:15.172] epoch 3, loss 0.0498
## INFO  [16:03:16.992] epoch 4, loss 0.0375
## INFO  [16:03:18.943] epoch 5, loss 0.0303
## INFO  [16:03:20.734] epoch 6, loss 0.0256
## INFO  [16:03:22.576] epoch 7, loss 0.0223
## INFO  [16:03:24.410] epoch 8, loss 0.0198
## INFO  [16:03:26.225] epoch 9, loss 0.0178
## INFO  [16:03:28.024] epoch 10, loss 0.0162
```

```
wv_context <- glove$components
local_glove <- wv_main + t(wv_context) # word vectors

# qualitative check
find_nns(local_glove['immigration',], pre_trained = local_glove, N = 5, candidates = feats)
```

```
## [1] "immigration" "bill"        "people"      "law"         "congress"
```

Estimating the transformation matrix

Given a corpus and it's corresponding GloVe embeddings, we can compute a corresponding transformation matrix using `conText::compute_transform()`.

```
# compute transform
# weighting = 'log' works well for smaller corpora
# for large corpora use a numeric value e.g. weighting = 500
# see: https://arxiv.org/pdf/1805.05388.pdf
local_transform <- compute_transform(x = toks_fcm, pre_trained = local_glove, weighting = 'log')
```

You should always run some sanity checks to make sure the computed transformation matrix produces sensical results (keep in mind the corpus in this example consists of only 200 documents). A more robust approach (not illustrated below) to verify your transformation matrix is to randomly select a set of features to exclude from the its estimation but which have a corresponding pre-trained embedding. Use the transformation matrix and pre-trained embeddings to compute ALC embeddings for these excluded features and check their similarity with their corresponding pre-trained embeddings (it should be reasonably high, > 0.6).

```
#---------------------------------
# check
#---------------------------------

# create document-embedding matrix using our locally trained GloVe embeddings and transformation matrix
immig_dem_local <- dem(x = immig_dfm, pre_trained = local_glove, transform = TRUE, transform_matrix = l

# take the column average to get a single "corpus-wide" embedding
immig_wv_local <- colMeans(immig_dem_local)

# find nearest neighbors for overall immigraiton embedding
find_nns(immig_wv_local, pre_trained = local_glove, N = 10, candidates = immig_dem_local@features)
```

```
##  [1] "immigration" "bill"        "people"      "law"         "reform"
##  [6] "illegal"     "can"         "system"      "president"   "country"
```

```
# we can also compare to corresponding pre-trained embedding
sim2(x = matrix(immig_wv_local, nrow = 1), y = matrix(local_glove['immigration',], nrow = 1), method =
```

```
##           [,1]
## [1,] 0.7849437
```

Other

Feature embedding matrix

We may be interested in simultaneously embedding many features. This can be useful to (1) compare two corpora along many features simultaneously and/or (2) estimate context-specific feature embeddings that can subsequently be fed into a downstream classification task. We'll focus on (1) here. Using the same tokenized `cr_sample_corpus` we start by building a feature-co-occurrence matrix for each group (party).

```
# create feature co-occurrence matrix for each party (set tri = FALSE to work with fem)
fcm_D <- fcm(toks[docvars(toks, 'party') == "D",], context = "window", window = 6, count = "frequency",
fcm_R <- fcm(toks[docvars(toks, 'party') == "R",], context = "window", window = 6, count = "frequency",
```

Given an `fcm` for each group, we can proceed to compute a corresponding "feature-embedding-matrix" for each group.

```
# compute feature-embedding matrix
fem_D <- fem(fcm_D, pre_trained = cr_glove_subset, transform = TRUE, transform_matrix = cr_transform, v
fem_R <- fem(fcm_R, pre_trained = cr_glove_subset, transform = TRUE, transform_matrix = cr_transform, v

# cr_fem will contain an embedding for each feature
fem_D[1:5,1:3]
```

```
## 5 x 3 sparse Matrix of class "dgCMatrix"
##
## president 0.3744276 0.2664312 0.5106154
## rise      0.6826160 0.1579899 0.1132372
## today     0.5012829 0.1986229 0.1827530
## honor     0.6683875 0.3323882 0.2093607
## one       0.5223592 0.4007449 0.2207826
```

Finally, we can use the `conText::feature_sim` function to compute "horizontal" cosine similarities between both `fem`s for the set of overlapping features. The output of `feature_sim` is ranked from least similar to most similar features.

```
# compute "horizontal" cosine similarity
feat_comp <- feature_sim(x = fem_R, y = fem_D)

# least similar features
head(feat_comp)
```

```
##       feature      value
## 1        link -0.2160803
## 2  guantanamo -0.1859814
## 3 frustration -0.1853331
## 4    opposite -0.1807265
## 5        draw -0.1754296
## 6     refuses -0.1743808
```

```
# most similar features
tail(feat_comp)
```

```
##          feature     value
## 3501     customs 0.9514645
## 3502      border 0.9520096
## 3503  department 0.9529625
## 3504    homeland 0.9562521
## 3505 enforcement 0.9570831
## 3506    security 0.9584692
```

Embedding full (short) documents

In all the examples above we've focused on contexts defined as "windows" around a target word. In Rodriguez et al. (2021) we show that the same procedure works well on contexts defined as the full document, as long as the full document is relatively short –exactly what threshold to use to define "short" is still an open question. Responses to open-ended survey questions is a good example (discussed in the paper). Below we provide an illustrative example using a subset of the `cr_sample_corpus`. In particular, note the use of the `. ~` operator to indicate that the full document should be used as the DV.

```
# identify documents with fewer than 100 words
short_toks <- toks[sapply(toks, length) <= 100,]

# run regression on full documents
model3 <- conText(formula = . ~ party,
                  data = short_toks,
                  pre_trained = cr_glove_subset,
                  transform = TRUE, transform_matrix = cr_transform,
                  bootstrap = TRUE, num_bootstraps = 10,
                  permute = TRUE, num_permutations = 100,
                  window = 6, case_insensitive = TRUE,
                  verbose = FALSE)
```

```
##   coefficient normed.estimate  std.error p.value
## 1     party_R       0.7175612 0.06838156    0.03
```