kNN-Suche syntaktischer Duplikate in Nachrichten-Korpora

Jonathan Schlue

Binning und Hashing von großen Vektoren mit FAISS

Abstract

Die Aufgabe besteht in dem Aufdecken von syntaktischen Duplikaten in einem Korpus von Sätzen. Mithilfe eines Word-Embeddings wird ein Sentence-Embedding von Sätzen in den mehrdimensionalen Raum konstruiert, aus dem wir mit *FAISS* (Facebook AI Similarity Search, [1]) einen kNN-Graphen generieren, in dem die Duplikate leicht abzulesen sind.

Vorbereitung

Zusätzliche Bibliotheken

Für die Vorverarbeitung in *R* verwenden wir zusätzliche Packete.

```
library(tm)
library(stringi)
library(proxy)
library(slam)
```

Vorverarbeitung

Einlesen

Wir lesen die Sätze aus der gebenen Datenmenge ein.

```
data <- read.csv(
  sentences.file,
  header=F,
  sep ='\t',
  stringsAsFactors = FALSE,
  quote = ""
)
sentences <- data[,2]
```

Bereinigung des Korpus'

Satzzeichen und überflüssige Leerzeichen werden entfernt. Zahlen entfernen wir auch, um Fehler beim Vergleich von Datum-Strings zu vermeiden. Stopwörter entfernen wir nicht, da sonst sehr kurze, aber unterschiedliche Sätze als Duplicate angesehen werden.

```
corpus <- VCorpus(VectorSource(sentences))
corpus <- tm_map(corpus, removePunctuation)
corpus <- tm_map(corpus, removeNumbers)
corpus <- tm_map(corpus, content_transformer(function(x) tolower(x)))
corpus <- tm_map(corpus, stripWhitespace)
```

Word-Embedding

Wir lesen das Korpus zunächst in eine Dokument-Term-Matrix mit Tf-Gewichten ein. Daraus können wir die maximale Satzlänge als untere Schranke für die Dimensionalität unserer Feature-Vektoren ableiten.

```
dtm <- DocumentTermMatrix(corpus, control=list(weighting=weightTf))
d <- max(row_sums(dtm)) + 1
num.terms <- length(dtm$dimnames$Terms)
num.docs <- length(dtm$dimnames$Docs)
```

Als naives Embedding ersetzen wir alle Terme im Korpus durch ihre korrespondierende Term-ID. Dafür erstellen wir zunächst ein Environment, dass sich wie eine Hashtabelle verhält.

```
term.id.map <- new.env()

for(i in 1:num.terms) {
  term <- dtm$dimnames$Terms[i]
  term.id.map[[ term ]] <- i
}

get.term.id <- function(term) {
  return(term.id.map[[term]])
}
```

Wir füllen die entsehenden Vektoren hinten mit dem Wert `-num.terms` auf. Dadurch wird später der Abstand der Features in den so aufgefüllten Dimensionen maximal zu den Features längerer Sätze.

```
term.ids <- lapply(as.list(corpus), function(doc) {
  content <- doc$content
  terms <- unlist(stri_split(content, regex = " "))
  terms <- intersect(terms, dtm$dimnames$Terms)
  term.ids <- lapply(terms, get.term.id)
  term.ids[(length(term.ids)+1):d] = -num.terms
  return(term.ids)
})
```

Schließlich befüllen wir unsere Feature-Matrix zeilenweise mit den Feature-Vektoren und normieren die enthaltenen Term-IDs.

```
xb <- matrix(unlist(term.ids), nrow = num.docs, byrow = T)
xb <- xb / num.terms
```

Die resultierende Matrix hinterlegen wir für die kNN-Suche mit FAISS.

```
write(t(xb), paste(opt$output, "xb.txt", sep="/"), d, sep = "\t")
```

Insgesamt sind die Terme alphabetisch geordnet, also ist die Differenz von Termen gerade der Abstand in der üblichen lexikographischen Ordnung. Es sei angemerkt, dass die Dimensionalität der Vektoren nun nicht von der Datenmenge sondern nur von der maximalen Satzlänge abhängt, die ihrerseits nicht von der Größe des Korpus abhängt.

Auswahl der zu vergleichenden Such-Indizes

Für die Auswahl der Indizes fallen folgende Kriterien an:

1. Korrektheit
2. Vollständigkeit
3. Speicherkomplexität
4. Datenmenge

Such-Indizes

Unter Betrachtung der der Richtlinien für die Auswahl von Indizes [2] können wir folgene Beobachtungen treffen: Die Indizes

- `IndexFlatL2` (Euklidische Distanz)
- `IndexFlatIP` (Skalarprodukt)

arbeiten vollständig und korrekt und Unterscheiden sich auschließlich in der Abstandsberechnung. Hinsichtlich Speicherverbrauch können wir folgende Einordnung treffen: Die Speicheranforderung der beiden Indizes ist `4*d Byte` pro Vektor, wobei `d` die Dimensionalität der Vektoren und wie bereits erläutert als konstant anzunehmen ist. Damit liegt eine Speicherkomplexität von `O(M)` vor, wobei `M` gerade die Kardinalität der Menge der Vektoren, also die Anzahl der Sätze im Korpus ist.

Damit lässt sich der Speicherverbrauch zur Laufzeit wie folgt Abschätzen:

```
4 * d Byte * M  < 4 * 30 * 1000000 Byte < 120 MB
```

Ergebnisse

Wir lesen die Ergebnisse, also die Nachbar-Ids und die Abstände in Matrizen ein. Dabei beachte man die Indexverschiebung von *Python* nach *R*.

```
I <- read.csv(
  paste('../out', "I.txt", sep = "/"),
  header=F,
  sep ='\t',
  stringsAsFactors = FALSE,
  quote = ""
)

D <- read.csv(
  paste('../out', "D.txt", sep = "/"),
  header=F,
  sep ='\t',
  stringsAsFactors = FALSE,
  quote = ""
)

k <- opt$neighbours

I <- matrix(unlist(I), ncol = k)
D <- matrix(unlist(D), ncol = k)
```

```
I <- I + 1
n <- nrow(I)
```

Als obere Schranke für den Abstand von Duplikaten wählen wir den höchsten Eigenabstand unter den Feature-Vektoren.

```
threshold <- max(D[,1])
```

Aus der Ergebnis-Abstands-Matrix lesen wir nun zunächst aus, welche Nachbarn Duplikate sind.

```
duplicate.neighbour.ids <- apply(D, 1, function(row) which(row <= threshold))
```

Darüber können wir auf die tatsächlichen Sätze zurückschließen.

```
get.duplicate.ids <- function(sentence.id) {
  duplicate.ids <- I[sentence.id, unlist(duplicate.neighbour.ids[sentence.id])]
  return(unlist(duplicate.ids))
}

duplicates <- sapply(1:n, get.duplicate.ids)
```

Wegen der Reflexivität der Ähnlichkeitsbeziehung betrachten wir nur die echten Duplikate, also alle Äquivalenzklassen mit Kardinalität echt größer eins.

```
real.duplicates <- duplicates
names(real.duplicates) <- 1:n
real.duplicates <- real.duplicates[sapply(real.duplicates, length) > 1]
```

Duplikate

Durch die Erhaltung der Satz-IDs können wir die Duplikate nun aus der Eingabedatenmenge ablesen.

```
result <- lapply(real.duplicates, function(duplicate_list) {
  result <- sentences[duplicate_list]
  names(result) <- duplicate_list
  return(result)
})
```

Schlussbetrachtung

Zusammenfassung

- Im Fall der Suche syntaktischer Duplikate können wir durch Kenntnisse über unseren Korpus bereits ein sehr vereinfachendes Word-Embedding verwenden
- Dadurch erhalten wir eine konstante und von der größe des Korpus' unabhängige Dimensionalität der Feature-Vektoren.
- Nun wird es einfach, unsere Kriterien an Speicherkomplexität auch bei exakter kNN-Suche zu erfüllen.

Locality-Sensitive Hashing

Nicht zuletzt soll an dieser Stelle der Index `IndexLSH` für *Locality-Sensitive Hashing* beziehungsweise *Localized Minimum Hashing* beleuchtet werden.

Beim LSH werden die eingegeben Feature-Vektoren zum Zweck der Dimensionsreduktion in binäre Feature-Vektoren gegebener Länge `n_bits` übersetzt, sodass ähnliche Ausgangsvektoren demselben binären Bildvektor zugeordnet werden. Dieses Verfahren nennt sich auch *Binning*, und der enstehende Bildvektor funktioniert wie ein Repräsentant für die Äquivalenzklassen bezüglich der Ähnlichkeitsbeziehung, etwa wie beim *Hashing*.

Da wir das Kriterium der Dimesnionalität bereits durch das Word-Embedding eliminiert haben, ist dieses Verfahren für die Suche nach syntaktischen Duplikaten irrelevant.

Qualtitative Vergleiche werden daher erst bei Datenmengen größerer Feature-Dimensionalität interessant. Es gilt dann, die Dimensionsreduktion vom *LSH* den Quantisierungsmethoden von *FAISS* unter den Kriterien

1. Korrektheit
2. Vollständigkeit
3. Speicherkomplexität
4. Datenmenge

gegenüberzustellen und die unterschiedlichen Verfahren quantitativ auszuwerten.

Ausblick

Die Fragestellung kann hinsichtlich semantischer Duplikate erweitert werden. Offenbar reicht ein wie in dieser Arbeit konstruiertes naives Word-Embedding nicht mehr aus. Sinnvoll wäre möglicherweise eine Kombination von Kookkurenzen, Satz-Kookkurenzen und einem POS-Tagging für die Erstellung der semantischen Feature-Vektoren. Die damit schier große Dimesionalität der Vektoren rückt das Problem der Dimensionsreduktion in ein neues Licht und macht die oben kurz angeschnittene, quantitative Betrachtung verschiedener Verfahren interessant.

Anhang

```
# ./bin/run.py

import faiss
import numpy as np
import csv
import sys

# parse CLI arguments

if len(sys.argv) == 3:
    out_dir = sys.argv[1]
    k = int(sys.argv[2])
else:
    out_dir = "../out"
    k = 2


# I/O functions

def csv_read(fname):
    reader = csv.reader(open(fname, "r"), delimiter="\t")
    x = list(reader)
    return np.array(x).astype("float32")
```

```
def csv_write(fname, data):
    writer = csv.writer(open(fname, "w"), delimiter="\t")
    writer.writerows(data)

# data processing

# extract
xb = csv_read("{}/xb.txt".format(out_dir))
d = xb.shape[1]

# transform

# build the index
index = faiss.IndexFlatL2(d)
print(index.is_trained)

# add vectors
index.add(xb)
print(index.ntotal)

# sanity check
D, I = index.search(xb[:5], k)
print(I)
print(D)

# actual search
D, I = index.search(xb, k)
print(I[:5])
print(I[-5:])

# load
csv_write("{}/D.txt".format(out_dir), D)
csv_write("{}/I.txt".format(out_dir), I)
```

Referenzen

1. https://github.com/facebookresearch/faiss/wiki/
2. https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index