DATA IN MOTION

Text Classification Studies

28.07.2023

Text Classification Studies

Part Contents

Section Contents

Introduction

- Our goal is to identify whether a certain field in a model may constitute a privacy concern under certain standards (e.g. GDPR);
- The suggestions should be based either on the field name or on the field description;
- We can see the problem as a one-class text classification problem, where our only class is the category of privacy-related terms, while all the other terms have to be identified as outliers;
- We can then train a model on a set of terms which we consider privacy-related, and then test it over some other examples.

Introduction

There are, however, several steps that have to be considered and understood before setting up such a model:

- **Preprocessing**: data has to be preprocessed before feeding a text classification model;
- **Tokenization**: the input text has then to be tokenized, maning divided in a set of tokens, containing only the relevant words (e.g. eliminating the stop words);
- **Vectorization**: this step consists in converting the text into a numerical representation, namely a vector, which is interpretable by the computer.

Introduction

Once we have figured the previous steps out, we can start building our model. Also for that we tried two approaches:

- A One-Class Text Classifier algorithm, provided by the python library scikit-learn;
- A simpler but more “under our control” approach, based on the computation of the cosine distance between the vector representations of the input text.

Results are then provided in terms of overall model accuracy, precision and recall.

Section Contents

Data Sample

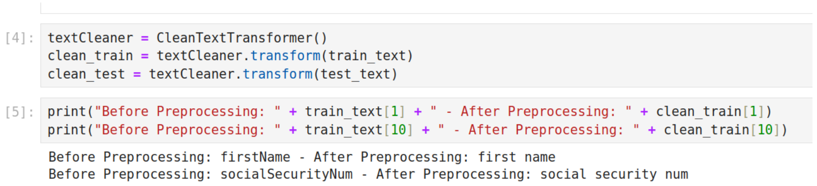
- As first thing, we load our train and test data sets;
- They consist of simple tables with a specified category (`IDENTITY` or `NO-RELATED` for privacy-related terms and non, respectively);
- The train set only consists of terms belonging to the first category, as this will be a one-class text classification problem. The test set, instead, contains both kinds of terms.

Preprocessing

In this step we would need to pass from all the possible forms in which our input text can be presented to a uniform and coherent form:

- Variable names have to be converted to "standard" text (e.g. `firstName` should become `first name`);
- Special symbols, such as `-`, $*$ have to be removed;
- Words have to be lemmatized (e.g. `places` should become `place`).

Preprocessing

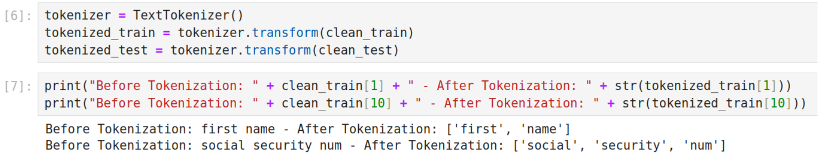


```
[4]: textCleaner = CleanTextTransformer()
     clean_train = textCleaner.transform(train_text)
     clean_test = textCleaner.transform(test_text)

[5]: print("Before Preprocessing: " + train_text[1] + " - After Preprocessing: " + clean_train[1])
     print("Before Preprocessing: " + train_text[10] + " - After Preprocessing: " + clean_train[10])

     Before Preprocessing: firstName - After Preprocessing: first name
     Before Preprocessing: socialSecurityNum - After Preprocessing: social security num
```

Tokenization

- Our data sample, after pre-processing, is just a list of strings, each string representing one document;
- What we want to achieve, instead, is to have a list of word tokens for every document;
- E.g. `["first name", "social security num"]` should become `[["first", "name"], ["social", "security", "number"]]`.

Tokenization



```
[6]: tokenizer = TextTokenizer()
     tokenized_train = tokenizer.transform(clean_train)
     tokenized_test = tokenizer.transform(clean_test)
```

```
[7]: print("Before Tokenization: " + clean_train[1] + " - After Tokenization: " + str(tokenized_train[1]))
     print("Before Tokenization: " + clean_train[10] + " - After Tokenization: " + str(tokenized_train[10]))
```

```
Before Tokenization: first name - After Tokenization: ['first', 'name']
Before Tokenization: social security num - After Tokenization: ['social', 'security', 'num']
```

Vectorization

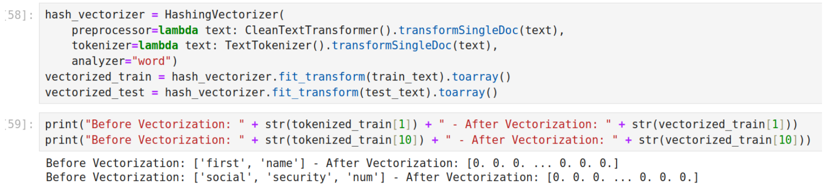
- The vector of tokens has to be converted into a vector of numbers;
- We would need some kind of representation of words into a numerical space;
- Ideally, words that are similar or analogous to each other would have a similar numerical representation, meaning, the distance between their vectors should be smaller compared to the one between two unrelated words.

Vectorization

There are different kinds of vectorization mechanisms, so we have to explore them a bit and to evaluate their performances for our use case. We have considered:

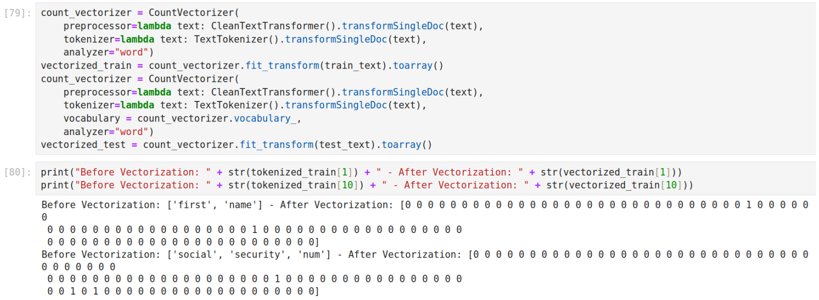
- *HashingVectorizer*
- *CountVectorizer*
- *GloVe Embedding*

Hashing Vectorizer



```
58]:  hash_vectorizer = HashingVectorizer(
          preprocessor=lambda text: CleanTextTransformer().transformSingleDoc(text),
          tokenizer=lambda text: TextTokenizer().transformSingleDoc(text),
          analyzer="word")
      vectorized_train = hash_vectorizer.fit_transform(train_text).toarray()
      vectorized_test = hash_vectorizer.fit_transform(test_text).toarray()

59]:  print("Before Vectorization: " + str(tokenized_train[1]) + " - After Vectorization: " + str(vectorized_train[1]))
      print("Before Vectorization: " + str(tokenized_train[10]) + " - After Vectorization: " + str(vectorized_train[10]))

      Before Vectorization: ['first', 'name'] - After Vectorization: [0. 0. 0. ... 0. 0. 0.]
      Before Vectorization: ['social', 'security', 'num'] - After Vectorization: [0. 0. 0. ... 0. 0. 0.]
```

Count Vectorizer

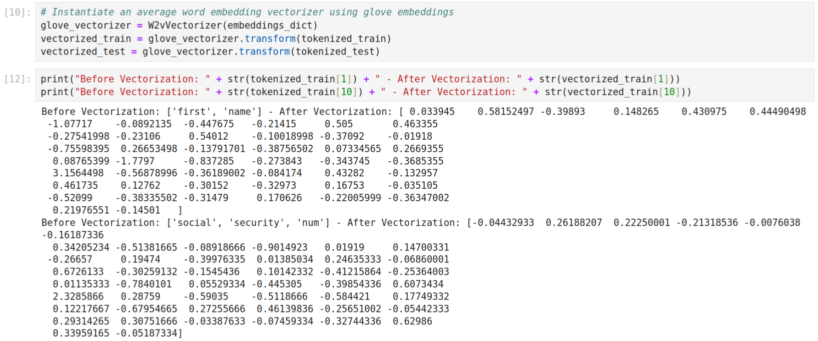


```
[79]: count_vectorizer = CountVectorizer(
          preprocessor=lambda text: CleanTextTransformer().transformSingleDoc(text),
          tokenizer=lambda text: TextTokenizer().transformSingleDoc(text),
          analyzer="word")
      vectorized_train = count_vectorizer.fit_transform(train_text).toarray()
      count_vectorizer = CountVectorizer(
          preprocessor=lambda text: CleanTextTransformer().transformSingleDoc(text),
          tokenizer=lambda text: TextTokenizer().transformSingleDoc(text),
          vocabulary = count_vectorizer.vocabulary_,
          analyzer="word")
      vectorized_test = count_vectorizer.fit_transform(test_text).toarray()
```

```
[80]: print("Before Vectorization: " + str(tokenized_train[1]) + " - After Vectorization: " + str(vectorized_train[1]))
      print("Before Vectorization: " + str(tokenized_train[10]) + " - After Vectorization: " + str(vectorized_train[10]))
```

```
Before Vectorization: ['first', 'name'] - After Vectorization: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
Before Vectorization: ['social', 'security', 'num'] - After Vectorization: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

GloVe Embedding



```
[10]: # Instantiate an average word embedding vectorizer using glove embeddings
      glove_vectorizer = W2vVectorizer(embeddings_dict)
      vectorized_train = glove_vectorizer.transform(tokenized_train)
      vectorized_test = glove_vectorizer.transform(tokenized_test)
```

```
[12]: print("Before Vectorization: " + str(tokenized_train[1]) + " - After Vectorization: " + str(vectorized_train[1]))
      print("Before Vectorization: " + str(tokenized_train[10]) + " - After Vectorization: " + str(vectorized_train[10]))
```

```
Before Vectorization: ['first', 'name'] - After Vectorization: [ 0.033945    0.58152497 -0.39893     0.148265    0.430975    0.44490498
 -1.07717    -0.0892135  -0.447675   -0.21415     0.505       0.463355
 -0.27541998 -0.23106     0.54012    -0.10018998 -0.37092    -0.01918
 -0.75598395  0.26653498 -0.13791701 -0.38756502  0.07334565  0.2669355
  0.08765399 -1.7797     -0.837285   -0.273843   -0.343745   -0.3685355
  3.1564498  -0.56878996 -0.36189002 -0.084174    0.43282    -0.132957
  0.461735    0.12762    -0.30152    -0.32973     0.16753    -0.035105
 -0.52099    -0.38335502 -0.31479     0.170626   -0.22005999 -0.36347002
  0.21976551 -0.14501   ]
Before Vectorization: ['social', 'security', 'num'] - After Vectorization: [-0.04432933  0.26188207  0.22250001 -0.21318536 -0.0076038
-0.16187336
  0.34205234 -0.51381665 -0.08918666 -0.9014923   0.01919     0.14700331
 -0.26657     0.19474    -0.39976335  0.01385034  0.24635333 -0.06860001
  0.6726133  -0.30259132 -0.1545436   0.10142332 -0.41215864 -0.25364003
  0.01135333 -0.7840101   0.05529334 -0.445305   -0.39854336  0.6073434
  2.3285866   0.28759    -0.59035    -0.5118666  -0.584421    0.17749332
  0.12217667 -0.67954665  0.27255666  0.46139836 -0.25651002 -0.05442333
  0.29314265  0.30751666 -0.03387633 -0.07459334 -0.32744336  0.62986
  0.33959165 -0.05187334]
```

Section Contents

Model Evaluation

- We have built a model with all 3 of the vectorization procedures listed above;
- The data was previously preprocessed and tokenized as described every time;
- After vectorizing, we evaluated the performance of each procedure in two ways:
  - Cosine Distance between test and train sets;
  - One-Class Text Classification SVM model.

Cosine Distance

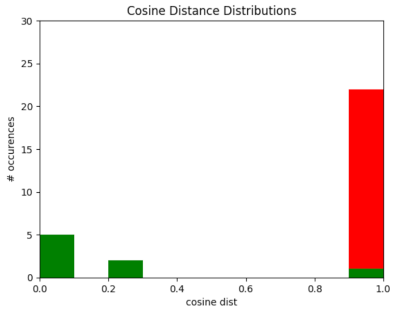
- For every document in the test set, we compute the minimum cosine distance between its vector representation and the ones of the documents in the training set;
- We then plotted the distributions of such distances for test documents which belong to the `IDENTITY` category (in green) and those which belong to the `NO-RELATED` category (in red).
- We then looped over 100 values of possible threshold, to find the optimal distance discriminator in order to optimize the accuracy of our model.

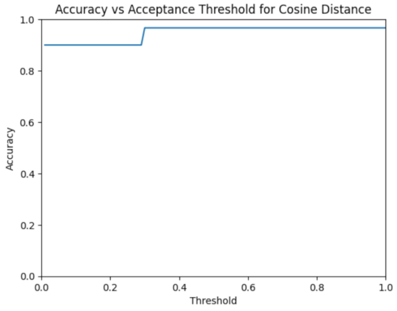
One-Class Text Classification SVM

- We built a One-Class SVM model using as input the preprocessed-tokenized-vectorized data;
- This model has 2 hyperparameters to be set;
- We looped over several values of these parameters, and took the ones that maximize the model accuracy;
- We then tested the model over the test set and compute the accuracy matrix.

Section Contents

Hashing Vectorizer - Cosine Distance



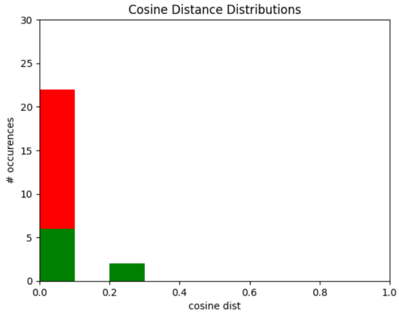


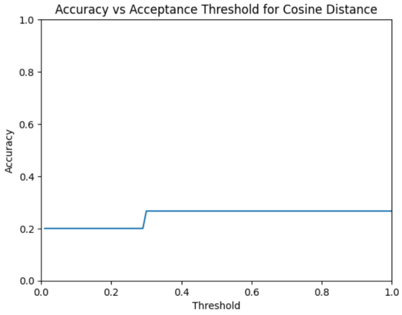
Precision	Recall	Accuracy
1.0	0.875	0.97

Hashing Vectorizer - OneClassSVM

Precision	Recall	Accuracy
1.0	0.75	0.93

Count Vectorizer - Cosine Distance



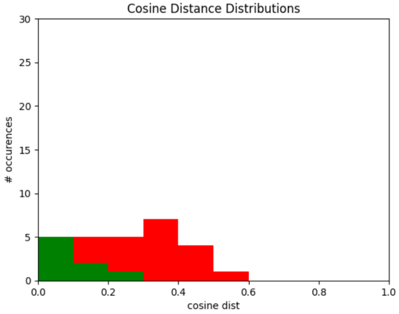


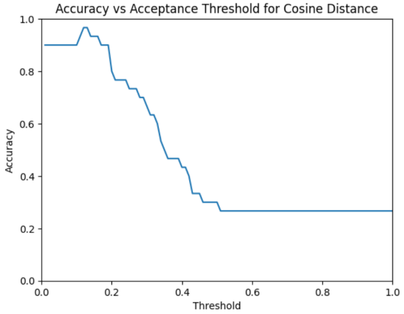
Precision	Recall	Accuracy
0.27	1.0	0.27

Count Vectorizer - OneClassSVM

Precision	Recall	Accuracy
0.19	0.62	0.17

GloVe Embedding - Cosine Distance





Precision	Recall	Accuracy
1.0	0.875	0.97

GloVe Embedding - OneClassSVM

Precision	Recall	Accuracy
1.0	0.38	0.83

Section Contents

Conclusions

- Cosine distance and OneClassSVM seems to perform in a similar way given the same vectorizer (which is a good indication we are doing something right);
- I personally prefer to use a cosine distance approach because I know what I am doing, while the OneClassSVM is a bit less under control, due to the hidden layer;
- The Count Vectorizer performs poorly, and this is probably due to the fact that it uses as vocabulary just the unique words found in the training set;
- GloVe has a well pre-trained model and we can test it also in other dimensions (currently we are using a vectorization with 50 dimensions, but there are also in 100D and more);
- Overall, I would go on with the GloVe approach and the cosine distance as model.

Next Steps

- Build a pipeline which takes as input the EStructuralFeature names and attribute descriptions and give back a suggestion based on such model on whether they might be privacy-related terms;
- Test the pipeline with some real ecore files;
- Provide a small UI to easily test everything;
- Provide the possibility to retrain the model when a new attribute which was not identified correctly is marked by the user.

Conclusion

Useful Links

OSGi Working Group

Working Group: www.osgi.org
WG Blog: www.osgi.org/blog
Twitter: @osgiwg
Bndtools: bndtools.org

Data In Motion

Web: www.datainmotion.com
Blog: datainmotion.com/blog
Twitter: @motion_data

Jürgen Albert

Email: j.albert@data-in-motion.biz

Mark Hoffmann

Email:
m.hoffmann@data-in-motion.biz