String Subsequence Kernel (SSK) for text classification

A study and re-implementation based on the original paper *Text Classification using String Kernels*, by Lodhi, Saunders, Shawe-Taylor, Christianini and Watkins [1]

Margarita Geleta

November 29, 2019

Introduction

The aim of this project is to study and implement a non-standard kernel, that is, a kernel with input space $\mathcal{X} \neq \mathbb{R}^d$, in our case the elements of this space will be strings, in order to apply an Support Vector Machine (SVM) model to perform Sentiment Analysis upon texts. Sentiment Analysis is a classification task which categorizes the orientation of a text into either positive or negative. The problem we will be dealing in this report is the classification of phrases.

String Subsequence Kernel (SSK)

As cited in [1], standard learning systems operate on input data once they have been transformed into feature vectors living in an d-dimensional space, $\mathbb{R}^d$.

Nevertheless, there exist kinds of raw data which cannot be naturally described in terms of explicit feature vectors. To illustrate: images, graphs or text are clear examples of such data. An alternative to explicit feature extraction are *Kernel Methods*, based on specific functions called *kernels*, $k: \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, which return the inner product between the mapped data in a higher dimensional space [1]. A Support Vector Machine (SVM) classifier is a supervised learning method based on *Kernel Methods*. Because of kernels, a great of feature of SVMs is that they can efficiently handle data in high dimensional spaces [3]. They build a boundary to separate classes and search for the optimal separator hyperplane.

Apparently, text classification brings high dimensionality. Texts represent *dense concepts* and thus, it is difficult to explicitly extract feature vectors from them, but at the same time, the strings that compose a text are *sparse instances* in a high-dimensional space. This is the reason why SVMs are appropriate for text categorization [2].

The most difficult choice in all this process is to choose a suitable kernel function which will compute the similarity between two elements from input space $\mathcal{X}$. In our case, those elements are strings s, which are finite sequences of characters from alphabet Σ, i.e. $s \in \Sigma^*$ ($\mathcal{X} = \Sigma^*$) strings are elements of the space of all subsequences of arbitrary length.

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$$

Clearly, $\Sigma^* \neq \mathbb{R}^d$.

A string kernel is a non−standard kernel for computing the inner product between two strings, $k: \Sigma^* \times \Sigma^* \to \mathbb{R}$. The approach proposed by [1] is formulated as follows: the more substrings of n−symbols two strings $s_1, s_2 \in \Sigma^*$ have in common, the more similar they are considered. Note that such substrings do not need to be contiguous and the degree of the contiguity of each substring will be weighted by a decay factor $\lambda \in (0,1]$. In contrast to the *Spectrum Kernel* (SK), which counts the occurences of a contiguous substrings $u \in \Sigma^n$ in s [4], the kernel proposed by *Lodhi et al.* does also capture the information of non−contiguous substrings, making it even more powerful. This kernel is the *String Subsequence Kernel* (SSK).

Let us introduce the formal definition of SSK with an example. Consider $s_1 = \text{man}, s_2 = \text{men}$, with $\Sigma = \{\text{a}, \text{e}, \text{m}, \text{n}\}$. And suppose $n = 2$, that is, we want to consider all the non−contiguous substrings of length 2. In s_1, we have substrings ma, an, and m_n, with _ denoting a gap. The substring with the gap will be weighted with λ accordingly to the length of the gap. The feature mapping ϕ for a string s measures the number of occurences of subsequences $u \in \Sigma^n$ in s weighting them according to their lengths (with a decay for gaps):

$$\phi_{u\in\Sigma^n}(s) = \sum_{\boldsymbol{i} \mid u=s[\boldsymbol{i}]} \lambda^{|\boldsymbol{i}|}$$

Where $|\boldsymbol{i}| = \left|\{i_1, \dots, i_{|u|}\}\right|$ the set of indices with $1 \le i_1 < \cdots < i_{|u|} \le |s|$, such that $u_j = s_{i_j}$, for $j = 1, \dots |u|$, or $u = s[\boldsymbol{i}]$ for short). Note that $|\boldsymbol{i}| = i_{|u|} - i_1 + 1$. The generated feature space is $\subseteq \mathbb{R}^{\binom{|\Sigma|}{n}}$.

Back to our example, $\phi_{\text{ma}}(s_1) = \lambda^2$ ($|\text{ma}| = 2$) and $\phi_{\text{mn}}(s_1) = \lambda^3$ ($|\text{m\_n}| = 3$). Thus, the inner product of two feature vectors for two strings s_1 and s_2 is computed by the sum over all common subsequences weighted according to their frequency of occurrence and lengths:

$$\begin{aligned} k_n(s_1, s_2) &= \sum_{u\in\Sigma^n} \langle \phi_u(s_1), \phi_u(s_2) \rangle \\ &= \sum_{u\in\Sigma^n} \sum_{\boldsymbol{i} \mid u=s_1[\boldsymbol{i}]} \lambda^{|\boldsymbol{i}|} \sum_{\boldsymbol{j} \mid u=s_2[\boldsymbol{j}]} \lambda^{|\boldsymbol{j}|} \\ &= \sum_{u\in\Sigma^n} \sum_{\boldsymbol{i} \mid u=s_1[\boldsymbol{i}]} \sum_{\boldsymbol{j} \mid u=s_2[\boldsymbol{j}]} \lambda^{|\boldsymbol{i}|+|\boldsymbol{j}|} \end{aligned}$$

We can compute the unnormalized kernel between s_1, s_2 by performing an inner product between their feature vectors:

	an	en	ma	me	mn
$\phi(s_1)$	λ^2	0	λ^2	0	λ^3
$\phi(s_2)$	0	λ^2	0	λ^2	λ^3

$$k_{n=2}(s_1, s_2) = \lambda^6$$

The normalized kernel is computed as:

$$\begin{aligned} \hat{k}_{n=2}(s_1, s_2) &:= \frac{k_{n=2}(s_1, s_2)}{\sqrt{k_{n=2}(s_1, s_1) \cdot k_{n=2}(s_2, s_2)}} \\ &= \frac{\lambda^6}{\sqrt{(2\lambda^4 + \lambda^6)^2}} = \frac{\lambda^2}{(2 + \lambda^2)} \end{aligned}$$

As $k_{n=2}(s_1, s_1) = k_{n=2}(s_2, s_2) = 2\lambda^4 + \lambda^6$. This may be feasible for small strings, but this computation is really impractical for moderate length texts: the feature vectors grow fast with Σ (the size is $\subseteq \mathbb{R}^{\binom{|\Sigma|}{n}}$) and a naive computation of SSK would involve $O(|\Sigma|^n)$ time and space [1]. A recursive approach with pruning and dynamic programming is capable of lowering the bound up to $O(n|s_1||s_2|)$ time. However, the computational cost is still high for large texts. Therefore, a SSK approximation was presented, based on a special case of an empirical kernel map introduced by Schölkopf et al. (1999). It consists in selecting not all substrings $u \in \Sigma^n$ but the ones occuring the most frequently in the dataset, based on the idea that those substrings are more likely to be highly informative. Neglecting the fact that there may be some loss of information, this yields a good approximation of SSK, reduces the feature space size and the overall complexity.

As you may have already noticed, the SSK has two hyperparameters: the length of subsequences n and the decay factor λ.

$\lambda =$	0.25	0.5	0.75	1
$\hat{k}_{n=2}(man, men)$	0.03	0.11	0.21	0.33
$\hat{k}_{n=2}(man, woman)$	0.70	0.67	0.62	0.54
$\hat{k}_{n=2}(man, human)$	0.70	0.67	0.62	0.54
$\hat{k}_{n=2}(man, women)$	0.02	0.07	0.13	0.18
$\hat{k}_{n=2}(woman, women)$	0.50	0.52	0.55	0.60
$\hat{k}_{n=2}(woman, human)$	0.49	0.46	0.39	0.30
$\hat{k}_{n=2}(women, human)$	0.01	0.05	0.08	0.10

As $\lambda \to 0$, the lesser importance we are giving to non-consequent subsequences since λ controls the penalization of the interior gaps of the substrings.

According to [1], the performance of the classifier varies with respect to varying the subsequence length n. Shorter or moderate non-contiguous substrings (of length n between 4 to 7) are capable of capturing the semantics better than longer non-contiguous substrings.

Data description

Our first try has been to deal with a dataset comprised of *tweets*, each one labeled with 0 and 1, meaning either negative or positive, respectively, obtained from *Kaggle*. However, we encountered some issues: first of all it required a heavy preprocessing to get rid of "@" and user names, which do not aport sentiment. Secondly, the number of tweets was huge and they were all of moderate length – the SSK required much time to build the Gram matrix. We did not have sufficient resources to run instances of SSK along with the SVM classifier. Thus, we have restricted ourselves to a smaller set of phrases, built by us.

Another point is that we first have tried to compute inner products between pieces of text. But after the training, the testing results were not satisfactory. Therefore we have switched to compute inner products between words that compound those texts. After that, a SVM classifier is trained with that set of words. When predicting a test sample, what we do is to split the text string into words; each word gets a prediction from our ready-made classifier; after that we can compute a % of positivity and negativity in the text, according to the prediciton given to each word. This way, the results have obtained much higher accuracy, recall and precision.

Materials

First of all, we have decided to implement the SSK in R. Yet we have observed a failure in R when accessing to the last position in a string during recursion. When timing SSK, with the same examples s_1, s_2 in R it was computed in average in 13.32 milliseconds, whereas in Python in 0.308 milliseconds. Why? Because accesing to the last position of a string in Python is straightforward, because it interprets strings as vectors of chars. This is not the case for R – in R you cannot index strings as vectors. There is a function `subsrt` to access the last position of a string. Using the same string as an example, we have timed the time of access in Python and R. The comparison is the following: in Python it takes 0.0052 milliseconds, whereas in R, in average, it takes 2.35 milliseconds. At this level it can seem irrelevant, but when using larger texts, the difference in time complexity gets very noticeable. That is the reason why we have switched to Python to implement SSK, although, the version in R is also attached.

We have implemented the recursive version of SSK in Python. But for moderate texts, it was quite slow, with the resources we had available. Thus, we have borrowed an SSK implementation from [5], but not from the last commit. The last version is implemented in Cython – we have used the last commit which was completely in Python. Even though it is slower than the cythonic version, it was enough for our purpose.

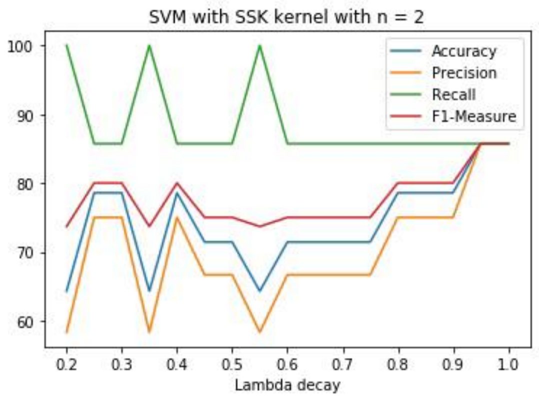
At this point the question was how to wrap our customized kernel into the SVM model from scikit library. We have struggled quite a time, until we have found a really disappointing answer in StackOverflow [6] – scikit's SVM implementation does not allow to get strings (or lists of strings) as training data. It does only allow to represent the strings in feature vectors – which actually we wanted to avoid with our string kernel. This was posted in 2014 and from that point there has not been any update on this issue. So we had no other option – we had to dig into the implementation of SVM and change that bug ourselves.

Methodology

Having the dynamic SSK and custom SVC ready, we passed to the next step – put them into practice. We have created a set of stemmed positive words, such as "`beaut`", "`kind`", "`good`" and positive suffixes, such as "`ful`", "`ness`", and saved them in an array called `positive`. Likewise, a set of stemmed negative words was created, with words such as "`dead`", "`ill`", "`bad`" and negative preffixes, such as "`anti`", "`dis`", "`un`", and stored them in an array called `negative`. Two arrays of labels were created, for positive words "`1`"s and for negative words "`-1`"s. Next, a training set was created concatening the above arrays. After shuffling the rows, we have instanced a *Support Vector Machine* classifier with SSK kernel defining the hyperparameters n and λ. Finally, it was trained with the training set and the model was ready for predicting.

Results

Small test sets have been created with positive and negative words which, either, did not exist in the training set, or were derivative terms of the existing ones, like "`beautiful`". We have started with $n = 2$, and we have used the test set to compute the accuracy, presicion, recall and F1-score measures to test the goodness of the classifier. The results are summarized in the following plot:



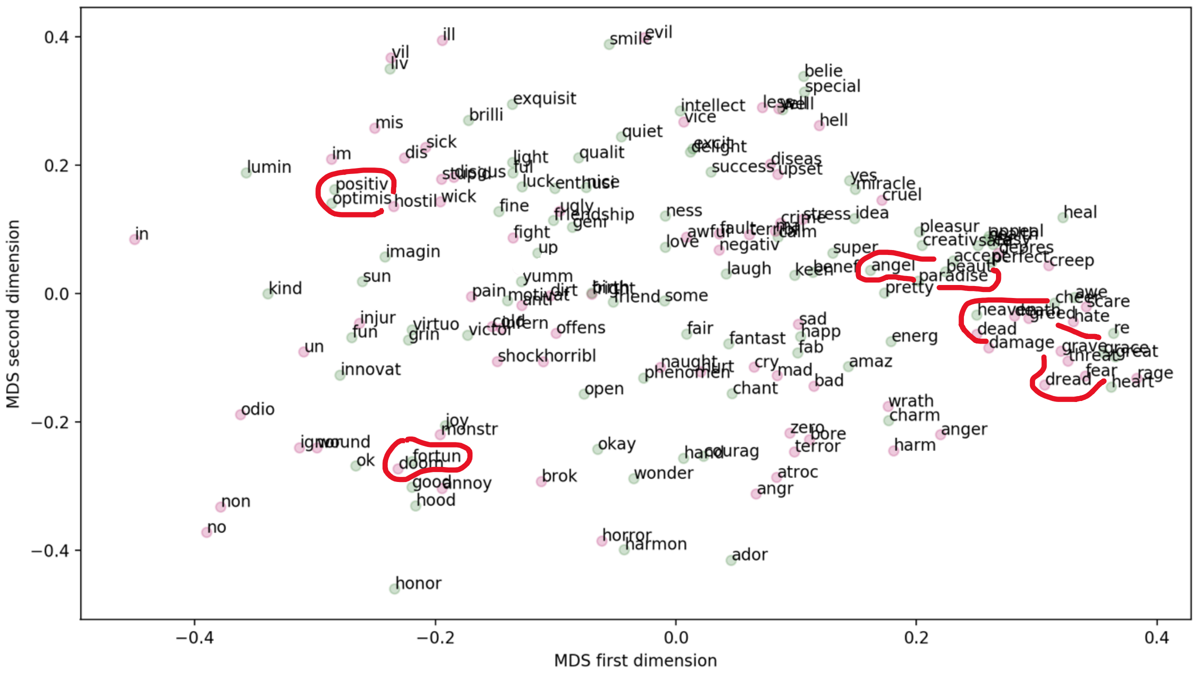
We have observed better results with higher values of λ, meaning that we obtain better results when we give more importance to non-consequent subsequences. This pattern was repeated for higher values of n ($n = 3, 4, \ldots$). Another feature we have observed is that with higher values of n, with $n = 3, 4$, we obtained higher values of accuracy, precision and F1-score for lower values of λ too. Which actually supports the empirical results shown in [1], larger substrings capture the semantics better. The maximum scores we have obtained:

Accuracy	85.71 %
Precision	85.71%
Recall	85.71%
F1-score	85.71 %

With $n = 2$ and $\lambda = 0.9, 1$, and also with $n = 4$ and $\lambda \in [0.7, \ldots, 1]$.

We have also implemented a function called `sentiment` which splits the input text into terms, cleans it from stopwords and classifies each term into either positive or negative. After that, it counts the number of words classified as positive and negative and outputs the percentage of positivity and negativity of the given text. Using $n = 4$ and $\lambda = 0.75$, we show some examples. The SVM-SSK classifier has determined that a birthday wish "`I hope your special day will bring you lots of happiness, love and fun. You deserve them a lot. Enjoy!`" has a sentiment of 72.73% positive. A wedding wish: "`May your joining together bring you more joy than you can imagine`" yielded a sentiment of 83.33% positive. A blessing: "`May you be filled with loving kindness. May you be well. May you be peaceful and at ease. May you be happy`" has a sentiment of 90.91% positive. Swearing "`Dammit you're an idiot`" has a sentiment of 100% negative. Another example: "`I had a terrible nightmare tonight`" has a sentiment of 66.67% negative.

Since SSK allows us computing the similarity matrix (the Gram matrix), to have a glimpse of how words are distributed in the feature space, we have tried to reduce the dimensionality of the space with Multi-Dimensional Scaling technique (MDS), previously computing the dissimilarity matrix subtracting the Gram matrix from the ones matrix. The plot of the words used in the training set is shown at the following page. We can observe that neighbour words are similar in terms of subsequences. For instance, `evil` and `smile` are similar according to SSK because both contain `il` and `e`. `Good`, `hood` and `doom` are similar because all of them contain the subsequence `oo` and are words comprised of four letters. Some of the words' clusters are quite curious, because they have semantic relationship (they are marked with red). There have been even studies



related to that field: for example, over a third of English profane words are four-letter words and others, longer profane words are just embeddings of the previous ones [7]. We have also computed de Gram and dissimilarity matrices of the *Full List of Bad Words and Top Swear Words banned by Google* [8]. The MDS plot showed quite an interesting result – obscene words appeared clustered according to their meaning. The plot is not shown in the report for the sake of appropriateness, many of the listed words are considered taboo expressions.

Discussion

As we can see, the results are quite promising. We have been able to obtain important measures like the *Accuracy* above 80% which having in mind the size of our train set is a very respectable percentage.

Our favourite part of this study is the utility of the method, as we have seen in the previous paragraphs – we can actually get a numerable measure of how positive or negative a phrase is. That could be of great interest for retail companies that have thousands of comments and reviews and want a general overview of how the public is reacting to a product.

And even though our method is implemented from scratch, so that it is not perfect: it can be improved and implemented more efficiently, the results are good enough for a small test. Then we could scale it up to whatever we would like or need.

This could also be useful for some social networks, if we have a child account or profile, we can protect it so that they do not see hateful or negative messages, in order to make the communication enjoyable without the "*dark side*" of networks like "Facebook" or "Twitter".

Conclusions

After all the work we can elaborate some final thoughts.

The kernel for strings idea was solid, it made sense but then the implementation was really difficult to accomplish. We spent more time trying to make it work than any other thing. But in the end we did get it to work and we are happy with the results so it evens out.

Once we have it implemented, we could really appreciate the potential of the method, and we have been able to get a better understanding of how it works and why it works that way.

We also learnt a lot by touching and modifying existing and already implemented methods, like the SVM, and merging it with our code, because we can modify every part, then try different things and perhaps try to implement some of the functions ourselves and see whether it works the way we think it should.

Future work

Having this project done, we imagine some ways to use this method and how to implement it in some useful ways.

We have commented it before, it can be useful for product overview and parental control. The *product overview* seems more reasonable with our current knowledge and *parental control* or other applications are a bit out of reach for us at the moment.

We actually found an Amazon dataset with thousands of reviews of products, and some papers documenting some works done with it. But the file was so big that only the preprocessing took nearly hours, so with the resources we had available, it was impossible to fit any SVMs, but it seemed like a viable thing in the near future.

References

[1] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, C. Watkins. Text Classification using String Kernels. Journal of Machine Learning Research 2 (2002) 419-444.

[2] T. Joachims. Text Categorization with Support Vector Machines: Learning with Many Relevant Features. In Claire Nédellec and Céline Rouveirol, editors, Proceedings of the European Conference on Machine Learning, pages 137 – 142, Berlin, 1998. Springer.

[3] scikit-learn. Support Vector Machines. https://scikit-learn.org/stable/modules/svm.html

[4] L. A. Belanche. Machine Learning II lecture notes. Polytechnical University of Catalonia.

[5] helq GitHub. Fast String Kernel (SSK) implementation for python. https://github.com/helq/python-ssk

[6] StackOverflow. How to use string kernels in scikit-learn? https://stackoverflow.com/questions/26391367/how-to-use-string-kernels-in-scikit-learn?noredirect=1&lq=1

[7] Wired. The Science of Swear Words. https://www.wired.com/2016/09/science-swear-words-warning-nsfw-af/

[8] GitHub. Full List of Bad Words and Top Swear Words Banned by Google. https://github.com/RobertJGabriel/Google-profanity-words