ML2R Coding Nuggets
Kernel PCA for Word Embeddings

Christian Bauckhage ⋆
Machine Learning Rhine-Ruhr
Fraunhofer IAIS
St. Augustin, Germany

ABSTRACT
We address the general problem of computing word embeddings and discuss a simple yet powerful solution involving intersection string kernels and kernel principal component analysis. We discuss the theory behind kernel PCA for word embeddings and present corresponding Python / NumPy code. Overall, we demonstrate that the whole framework is very easy to implement.

1 INTRODUCTION
Previously [2], we already pointed out that machine learning for language processing often requires vectorial representations of words. Such embeddings of words into Euclidean vector spaces are now commonly computed using skip gram- or transformer models which learn semantic representations [10, 12, 16, 18]. Yet, simpler syntactic representations have their merits, too [4, 7, 8, 13]. They facilitate morphologically rich language processing, cope with infrequent or out-of-vocabulary words, and can be trained on small corpora. Last but not least, the underlying learning algorithms are light weight and easy to implement.

This note once again demonstrates the latter. We have already seen that plain vanilla Python makes it easy to implement so called intersection string kernels [2]. Here, we implement NumPy code for kernel principal component analysis (kPCA) and discuss how it applies to the practical problem of computing word embeddings.

2 THEORY
Next, we first recap theory (and practice) of intersection string kernels. We then discuss kernel principal component analysis (kPCA) and how it applies to the practical problem of computing word embeddings.

2.1 Recap: Intersection String Kernels
An alphabet \( \mathcal{A} \) of size \(|\mathcal{A}| = m\) is a set of \( m \) symbols \( \{a_1, \ldots, a_m\} \). A string \( s \) of length \( l \) over an alphabet \( \mathcal{A} \) is an ordered sequence of symbols \( s[i] \in \mathcal{A} \)

\[
\]  

We also write \( s \in \mathcal{A}^* \) where \( \mathcal{A}^* \) is the set of all possible sequences (finite or infinite) of symbols in \( \mathcal{A} \). If a string has length \( l \), it is also an element of \( \mathcal{A}^l \subset \mathcal{A}^* \) which denotes the set of all strings of length \( l \) over \( \mathcal{A} \).

The multiset of \( n \)-grams of a string \( s \) is the multiset

\[
M_n(s) = \left\{ s[i] : \ldots : s[i + n - 1] \, \middle| \, 1 \leq i \leq l - n + 1 \right\}
\]

of all contiguous sub-strings of \( s \) of length \( n \). The multiplicity of an \( n \)-gram \( g \in M_n(s) \) counts how often \( g \) occurs in \( M_n(s) \) and is denoted by \( n_{M_n}(g) \).

When working with Python, simple lists are a natural choice for implementing multisets and corresponding plain vanilla Python code for computing the \( n \)-grams of a string is shown in Listing 1.

```python
import numpy as np
import numpy.linalg as la
from collections import Counter
```

Figure 1: A didactic vocabulary of 36 words.
The nature of strings

If \( \phi \) computes an inner product in some Hilbert space \( k \)

shown in Listing 3.

nor about computing minima. Correspondingly simple code is

defined in (4). This is because the operator \( d \) in (4) is not a substring of \( s \).

Correspondingly simple code is provided by the module \texttt{collections} which is part of the standard library. Corresponding code for computing an \( n \)-gram histograms of \( s \) is shown in Listing 2.

The size of the intersection of the \( n \)-gram histograms of two strings \( s_i \) and \( s_j \) can be formalized as

\[
k_n(s_i, s_j) = \sum_{g \in A^n} \min \left\{ h_{n,s_i}[g], h_{n,s_j}[g] \right\}
\]

and counts how many \( n \)-grams the two strings have in common.

Working with \texttt{Python}, it is again easy to compute the expression in (4). This is because the operator \& can intersect counters. We therefore neither have to worry about summing over the \( g \in A^n \) nor about computing minima. Correspondingly simple code is shown in Listing 3.

Finally, we recall that one can show that function \( k_n(\cdot, \cdot) \) in (4) is an instance of a Mercer kernel [2].

\[ \Phi = [\varphi_1 \cdots \varphi_N] \in \mathbb{R}^{\dim(\mathcal{H}) \times N} \]

For the time being, we will \textit{assume} that these data are centered. In other words, we will assume that their feature space mean vector is the vector of all zeros. Formally, we state this as

The latter is the case for the map \( \varphi: \mathcal{A}^n \rightarrow \mathcal{H} \) which we constructed in [2] in order to prove that \( k_n(s_i, s_j) \) is a Mercer kernel.

However, from the point of view of “pen-and-paper math”, we can still work with \( \varphi \) and \( \mathcal{H} \). Understanding them as abstract mathematical constructs allows us to perform abstract mathematical operations on them. This principle will guide our following discussion and, once that discussion reaches its conclusion, all occurrences of feature space vectors will be in form of inner products which can then be swapped for kernel evaluations. In other words, the Hilbert space math we study next is practically computable.

In language processing practice, we typically work with whole sets of strings or \textit{vocabularys} of words. According to what we just said, we can conceptualize such a vocabulary in terms of a set of feature vectors which we may gather in a matrix

\[ C = \frac{1}{N} \Phi \Phi^\top \]

and ask for its eigenvalues \( \lambda_r \) and feature space eigenvectors \( u_r \).

\[ C u_r = \lambda_r u_r \]

To see if these reveal structural properties or other insights into the nature of our data.

To tackle these feature space eigenvector / eigenvalue problems, we first note the equivalences

\[
\Phi \varphi_j = \lambda_r \mathbf{u}_r \quad \Phi \mathbf{v}_r = \lambda_r \mathbf{u}_r
\]

where

\[
\mathbf{u}_r \in \mathbb{R}^N, \quad \mathbf{v}_r \in \mathbb{R}^N
\]

Looking at (14), we observe that each eigenvector \( \mathbf{u}_r \) of \( \Phi \) is a linear combination of the columns \( \varphi_j \) of \( \Phi \). We also emphatically emphasize that \( \mathbf{u}_r \in \mathbb{R}^N \) whereas \( \mathbf{v}_r \in \mathbb{R}^N \).

Continuing from (14), we further have

\[ \Phi \mathbf{v}_r = \lambda_r \mathbf{u}_r \]

\[ \Phi^\top \Phi \mathbf{v}_r = \lambda_r \mathbf{u}_r \]

and note the crucial fact that \( \Phi^\top \Phi \) is an \( N \times N \) Gram matrix whose entries are given by

\[ (\Phi^\top \Phi)_{ij} = \varphi_i^\top \varphi_j \]
However, as the right hand side of (20) is the same as the right hand side of (7), we can now invoke the kernel trick and replace inner products by kernel evaluations

$$\Phi^\top \Phi$$

which can actually be implemented on a computer. In other words, the kernel trick allows us to leave the realm of purely conceptual math and enter the domain of practical computability.

In fact, we can introduce a whole kernel matrix $K \in \mathbb{R}^{N \times N}$ with entries

$$K_{ij} = k_\Phi(s_i, s_j)$$

and henceforth consider the eigenvector / eigenvalue problems

$$K v_r = \lambda_r v_r$$

Using software for numerical computing, these are easily solved, and, indeed, NumPy has us covered. But we need to discuss some more theory before we can dive into coding.

The important intermediate result at this point is that (23) allows for practically solving for the coefficient vectors $v_r \in \mathbb{R}^N$ which, according to (14), conceptually determine the sought after feature space eigenvectors $u_r \in \mathbb{H}$. However, we still must address several technical details.

First of all, using NumPy methods to solve (23) will produce eigenvectors $v_r$ which are normalized to unit length $\|v_r\| = 1$. But what we are actually after are feature space eigenvectors $u_r$ of length $\|u_r\| = 1$. We therefore need to re-normalize the $v_r$ that are produced by our software.

To infer the correct normalization, we left-multiply (14) by $u_r^\top$ to obtain $u_r^\top \Phi v_r = \lambda_r u_r^\top u_r$. Using (16), this can also be written as $v_r^\top v_r = \lambda_r u_r^\top u_r$. Since we want the $u_r$ to be unit vectors, we posit $u_r^\top u_r = 1$. But this is then to say that

$$\frac{1}{\lambda_r} v_r^\top v_r = 1$$

from which we deduce the required re-normalization, namely

$$v_r \leftarrow \frac{v_r}{\sqrt{\lambda_r}}$$

Note: It may happen that some of the eigenvalues of the kernel matrix $K$ are 0. In other words, if we assume that the eigenvalues are ordered descendingly $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_{N-1} \geq \lambda_N$, there may be an $\lambda_{p+1} > 0$ such that $\lambda_{p+1} = \ldots = \lambda_N = 0$. Practical implementations must pay attention to this potential issue in order to avoid (division by zero) runtime errors.

Second of all, to keep our equations clean, we assumed that matrix $\Phi$ was centered. While this will rarely be the case, a matrix which is guaranteed to be centered is $\Phi J$ where

$$J = I - \frac{1}{N} 11^\top$$

is the centering matrix we studied before [3]. From our discussion back then, we recall the following practically crucial insights: Replacing each occurrence of $\Phi$ in (19) by $\Phi J$, we obtain

$$J^\top \Phi^\top \Phi J v_r = \lambda_r v_r$$

which provided us with an “actionable” vector representation of string $s$, that is, one that does not live in some abstract or conceptual Hilbert space $\mathbb{H}$ but in the familiar Euclidean space $\mathbb{R}^d$.

At first sight, this may seem impractical because, as stressed above, we usually cannot compute with vectors $\varphi(s), u_r \in \mathbb{H}$.

Yet, once again, the kernel trick comes to our rescue. This is because equation (14) allows us to rewrite the feature space inner product $\varphi(s)^\top u_r$ as follows

$$\varphi(s)^\top u_r = \frac{1}{\lambda_r} \varphi(s)^\top \Phi v_r$$

Note: Division by $\lambda_r$ is dictated by (14). However, at this point this constitutes mere scaling and does not fundamentally impact the value of the inner product (by which we mean that it may scale.

2.3 Kernel PCA for Word Embeddings

All the above was possibly well known and straightforward, but now the crucial questions are: What can we do with the eigenvectors $v_r$ of our (centered) intersection string kernel matrix $K_c$? What are they actually good for?

Well, as teased in the introduction, a prime application of kernel PCA on string kernel matrices is the computation of vector space embeddings of the words in our vocabulary. To see how to proceed, we still need some more theory.

Given any string $s \in \mathcal{A}^*$, our general idea is to project feature space vectors $\varphi(s) \in \mathbb{H}$ onto feature space eigenvectors $u_r \in \mathbb{H}$ to obtain numbers $x_r(s) \in \mathbb{R}$. In other words, the idea is to compute

$$x_r(s) = \varphi(s)^\top u_r$$

Why? Because, if we did this for $r \in \{1, 2, \ldots, d\}$, we would obtain a real valued vector

$$x(s) = \begin{bmatrix} x_1(s) \\ x_2(s) \\ \vdots \\ x_d(s) \end{bmatrix}$$

which provided us with an “actionable” vector representation of string $s$.
its value but will leave its sign intact). We therefore simply drop this scaling and consider
\[ \varphi(s)^T \nu_r = \varphi(s)^T \Phi \nu_r \] (37)

Now, looking at the expression \( \varphi(s)^T \Phi \nu_r \), we realize that it involves inner products between the feature space vector \( \varphi(s) \) and the feature space vectors \( \varphi_1, \ldots, \varphi_N \) gathered in matrix \( \Phi \). But this is to say that we may write
\[ \varphi(s)^T \Phi \nu_r = k(s)^T \nu_r = \nu_r^T k(s) \] (38)
where the entries of the kernel vector \( k(s) \in \mathbb{R}^N \) are given by
\[ k(s)_j = \varphi(s)^T \varphi_j = k_n(s, s_j) \] (39)

In other words and in conclusion, an embedding of a string \( s \) into a vector space \( \mathbb{R}^d \) can be computed as
\[ x(s) = \nu_r^T k(s) \] (40)
where matrix
\[ V_d = [\nu_1 \cdots \nu_d] \in \mathbb{R}^{N \times d} \] (41)
contains the (properly re-normalized) \( k \) leading eigenvectors of the centered kernel matrix \( K_c \).

One final remark appears to be in order: If we want to compute (41) for the strings \( s_j \) in the vocabulary from which we computed the (original, i.e. non-centered) kernel matrix \( K \), we have hardly any work to do. This is because \( [k(s_i)]_r = \varphi_i^T \varphi_j \) so that \( k(s_i) \) is nothing but the \( i \)-th column of \( K \).

A matrix \( X \) whose columns represents embeddings of all the words in our vocabulary can thus be computed as easily as
\[ X = V_d^T K \] (42)

3 PRACTICE

This section shows how to implement the ideas we just expounded. To work with a practical example, we consider the vocabulary in Fig. 1 which we represent as a Python list of strings

\[
\text{VOC} = ["bouvier patty", \ldots, "wiggum ralph"]
\]

Given this vocabulary, we begin by computing a list of \( n \)-gram histograms. Opting for \( n = 3 \), this can be accomplished by means of
\[
\text{vocHists} = [\text{n_gram_hist(word, n)} \text{ for word in VOC}]
\]

Based on this list, we can compute a corresponding intersection string kernel matrix \( K \). For this, we call compute_kernel_matrix in Listing 5 as follows

```python
def compute_kernel_matrix(hs):
    matK = np.zeros((len(hs), len(hs)))
    for i, h_i in enumerate(hs):
        for j, h_j in enumerate(hs, i):
            matK[i,j] = np.sum((h_i & h_j).values())
            matK[j,i] = matK[i,j]
    return matK
```

we obtain a vector \( \tilde{\lambda} \) of ascending eigenvalues and a matrix \( V \) whose columns contain the corresponding eigenvectors \( \nu_r \). Zero eigenvalues and corresponding eigenvectors can be discarded via

```python
def compute_kernel_vector(h, hs):
    vecK = np.zeros(len(hs))
    for i, h_i in enumerate(hs):
        vecK[i] = np.sum(h & h_i).values()
    return vecK
```

Now, in order to be able to visualize our results, we will embed the words in our vocabulary into \( \mathbb{R}^2 \). That is, we will compute \( X = V_d^T K \). Here, we need to remember that \texttt{la.eigh} returns eigenvalues / eigenvectors in ascending order. Since we are interested in leading eigenvectors, we may therefore proceed as follows

\[
\text{dims} = (-1, -2)
\]
\[
\text{matX} = \text{matV}[:, \text{dims}] \cdot \mathbf{T} \cdot \text{matK}
\]

If we plot the data in the columns of array \( \text{matX} \) as 2D points (and annotate them with the words they represent), we obtain a picture such as shown in Fig. 2a. Looking at this figure, it becomes clear why we referred to our framework as a syntax oriented way of computing word embeddings: Syntactically similar words end up near to each other in the embedding space \( \mathbb{R}^2 \).

This was straightforward, wasn’t it? But we also claimed that our approach extends to out-of-vocabulary words. So, to convince ourselves that it really does, we observe that our vocabulary in Fig. 1 lacks many words one would expect to see in the domain it has been sampled from. Let us therefore consider the following list of out-of-vocabulary words

\[
\text{OOV} = ["flanders maude", "simpson abe", "van houten kirk"]
\]

and compute kernel vectors for its elements.

Reusing our previous code, i.e. reusing list \text{vocHists} of tri-gram histograms, we may get these vectors for any word in \text{OOV} using

\[
\text{hist} = \text{n_gram_hist(word, n)}
\]
\[
\text{vecK} = \text{compute_kernel_vector(hist, vocHists)}
\]
where function `compute_kernel_vector` is defined in Listing 6.

In fact, we should compute a whole matrix $Y$ whose columns represent 2D word embeddings of our out-of-vocabulary words. Given what we just discussed, this can be accomplished via

$$matY = np.zeros((3, 2))$$

```python
    for i, word in enumerate(OOV):
        hist = n_gram_hist(word, n)
        vecK = compute_kernel_vector(hist, vocHists)
        matY[i] = matV[:, dims].T @ vecK
```

$$matY = matY.T$$

If we plot the resulting embeddings of the words in OOV together with the previously computed embeddings of the words in VOC, we obtain a picture such as in Fig. 2b.

It is noticeable that the newly embedded OOV words end up close to lexically similar VOC words. This once again suggests that our approach is reasonable. Using intersection string kernels and kernel PCA, it is rather easy to map strings $s \in \mathcal{A}^n$ to vectors $x(s) \in \mathbb{R}^d$. The resulting vector space embeddings reflect syntactic commonalities among given words and can thus provide useful structural information for a variety of downstream tasks.

4 CONCLUSION

Natural language processing for intelligent document analysis is a central topic of our work in ML2R. While many applications demand the use of powerful neural nets [5, 6, 9, 11, 14, 15, 19–21], there also are situations where more light weight approaches are preferable [4, 7, 8, 13]. A central machine learning problem in most of these settings is (to learn) to represent words, sentences, paragraphs, or even larger texts in terms of numerical vectors for further analysis.

In this note, we demonstrated that computing such vector space embeddings does not have to be demanding or training time intensive. Indeed, working with string kernels and kernel PCA hardly requires any training at all. Still, the framework yields useful numerical representations which reflect lexical similarities. This may be beneficial when dealing with languages where constituents of words carry substantial grammatical information.

The practical examples presented in this note were hopefully compelling but, at the same time, so simple that they could be computed on the fly. It goes without saying that real world applications which deal with much larger vocabularies would need components for efficient data management. That is, words, their $n$-gram histograms and vector representations, as well as whole kernel matrices should be stored in data bases which allow for fast access and easy extensibility.

ACKNOWLEDGMENTS

This material was produced within the Competence Center for Machine Learning Rhine-Ruhr (ML2R) which is funded by the Federal Ministry of Education and Research of Germany (grant no. 01IS18038C). The authors gratefully acknowledge this support.

REFERENCES

C. Bauckhage