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 allows for the embedding of words into the space spanned by the feature space principal components of a given vocabulary. This is useful for various downstream processing tasks, allows for visual analytics of whole vocabularies, and seamlessly extends to out-of-vocabulary words.

In section 2, we first recall the gist of our earlier discussion of intersection string kernels. We then discuss the general ideas and mathematics behind kernel PCA and how it applies to the problem of computing vector space embeddings of words (or, if need be, of whole sentences, paragraphs, or texts). In section 3, we put this theory into practice and discuss *NumPy* code for kernel PCA and subsequent projections onto kernel principal components. Note that our discussion assumes that readers have already had some experience with applied linear algebra and are familiar with the notion and properties of the eigenvalues and eigenvectors of a (sample covariance) matrix.

As always, we also expect that our readers have worked with *Python* and *NumPy* [17] before. Those who would like to experiment with our code snippets need to

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

bouvier patty	muntz nelson
bouvier selma	nahasapeemapetilon apu
brockman kent	prince martin
burns charles montgomery	qumby joe
carlson carl	riviera dr. nick
chalmers gary	simpson bart
flanders ned	simpson homer
flanders rod	simpson lisa
flanders todd	simpson maggie
frink prof. john	simpson marge
gumbel barney	skinner agnes
hibbert dr. julius	skinner seymour
krabappel edna	smithers waylon
leonard lenny	syslack moe
lovejoy helen	van houten luann
lovejoy timothy	van houten milhouse
mann otto	wiggum clancy
moleman hans	wiggum ralph

Figure 1: A didactic vocabulary of 36 words.

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 $s[i] \in \mathcal{A}$

$$s = s[1] : s[2] : \ldots : s[l] \tag{1}$$

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

$$\mathcal{M}_n(s) = \Big\{ s[i] : \ldots : s[i+n-1] \;\Big|\; 1 \leq i \leq l-n+1 \Big\} \tag{2}$$

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

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.

Listing 1: computing the n-grams of a string s

```
def n_grams(s, n):
    return map(''.join, zip(*[s[i:] for i in range(n)]))
```

Listing 2: computing the n-gram histogram of a string s

```
def n_gram_hist(s, n):
    return Counter(n_grams(s, n))
```

Listing 3: computing the intersection kernel of stings s_i, s_j

```
def intersection_str_kernel(si, sj, n):
    hi = n_gram_hist(si, n)
    hj = n_gram_hist(sj, n)

    return sum((hi & hj).values())
```

The n-gram histogram $h_{n,s}$ of a string s over $\mathcal{A}$ can be though of as a discrete function

$$h_{n,s} : \mathcal{A}^n \rightarrow \mathbb{N} \tag{3}$$

such that $h_{n,s}[g] = m_{\mathcal{M}_n(s)}(g)$ and, by convention, $h_{n,s}[g] = 0$ if g is not a substring of s.

In *Python*, a natural data structure for implementing histograms over hashable objects (such as text strings) is the dictionary subclass `Counter` provided by the module `collections` which is part of the standard library. Corresponding code for computing an n-gram histograms of of string 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

$$k_n(s_i, s_j) = \sum_{g \in \mathcal{A}^n} \min\Big\{ h_{n,s_i}[g],\, h_{n,s_j}[g] \Big\} \tag{4}$$

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

Working with *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 $g \in \mathcal{A}^n$ 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].

2.2 Kernel PCA

If $k_n(s_i, s_j)$ is a Mercer kernel, then theory tells us that it implicitly computes an inner product in some Hilbert space $\mathbb{H}$. This means that there has to exist some feature map $\boldsymbol{\varphi} : \mathcal{A}^* \rightarrow \mathbb{H}$ which turns strings s_i and s_j into feature vectors

$$\boldsymbol{\varphi}_i = \boldsymbol{\varphi}(s_i) \tag{5}$$

$$\boldsymbol{\varphi}_j = \boldsymbol{\varphi}(s_j) \tag{6}$$

such that

$$k_n(s_i, s_j) = \boldsymbol{\varphi}_i^\intercal \boldsymbol{\varphi}_j \tag{7}$$

The crux is that we typically do not know any details about the nature of $\boldsymbol{\varphi}$ or $\mathbb{H}$. Or, even if we did , it may still be practically infeasible to implement Hilbert space feature vectors on a computer. The latter is the case for the map $\boldsymbol{\varphi} : \mathcal{A}^* \rightarrow \mathbb{H}$ 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 $\boldsymbol{\varphi}$ and $\mathbb{H}$. 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 *vocabularies* 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

$$\boldsymbol{\Phi} = \big[\boldsymbol{\varphi}_1 \cdots \boldsymbol{\varphi}_N\big] \in \mathbb{R}^{\dim[\mathbb{H}] \times N} \tag{8}$$

For the time being, we will *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

$$\tfrac{1}{N}\, \boldsymbol{\Phi}\, \mathbf{1} = \mathbf{0} \tag{9}$$

Note: We only make this assumption to keep our equations legible. In practice, it will rarely hold; but this is not an issue to worry about. As we shall see soon, there is an easy solution to the feature space centering problem.

Having conceptualized the feature space data matrix $\boldsymbol{\Phi}$, we can, again conceptually, consider the $\dim[\mathbb{H}] \times \dim[\mathbb{H}]$ feature space sample covariance matrix

$$\boldsymbol{C} = \tfrac{1}{N}\, \boldsymbol{\Phi}\boldsymbol{\Phi}^\intercal \tag{10}$$

and ask for its eigenvalues λ_r and feature space eigenvectors $\boldsymbol{u}_r$

$$\boldsymbol{C}\, \boldsymbol{u}_r = \lambda_r\, \boldsymbol{u}_r \tag{11}$$

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 equivalencies

$$\tfrac{1}{N}\, \boldsymbol{\Phi}\boldsymbol{\Phi}^\intercal \boldsymbol{u}_r = \lambda_r\, \boldsymbol{u}_r \tag{12}$$

$$\Leftrightarrow \quad \boldsymbol{\Phi}\boldsymbol{\Phi}^\intercal \boldsymbol{u}_r = \tilde{\lambda}_r\, \boldsymbol{u}_r \tag{13}$$

$$\Leftrightarrow \quad \boldsymbol{\Phi}\, \boldsymbol{v}_r = \tilde{\lambda}_r\, \boldsymbol{u}_r \tag{14}$$

where

$$\tilde{\lambda}_r \equiv N\, \lambda_r \tag{15}$$

$$\boldsymbol{v}_r \equiv \boldsymbol{\Phi}^\intercal \boldsymbol{u}_r \tag{16}$$

Looking at (14), we observe that each eigenvector $\boldsymbol{u}_r$ of $1/N\, \boldsymbol{\Phi}\boldsymbol{\Phi}^\intercal$ is a linear combination of the columns $\boldsymbol{\varphi}_j$ of $\boldsymbol{\Phi}$. We also emphatically emphasize that $\boldsymbol{u}_r \in \mathbb{H}$ whereas $\boldsymbol{v}_r \in \mathbb{R}^N$.

Continuing from (14), we further have

$$\boldsymbol{\Phi}\, \boldsymbol{v}_r = \tilde{\lambda}_r\, \boldsymbol{u}_r \tag{17}$$

$$\Leftrightarrow \quad \boldsymbol{\Phi}^\intercal \boldsymbol{\Phi}\, \boldsymbol{v}_r = \tilde{\lambda}_r\, \boldsymbol{\Phi}^\intercal \boldsymbol{u}_r \tag{18}$$

$$\Leftrightarrow \quad \boldsymbol{\Phi}^\intercal \boldsymbol{\Phi}\, \boldsymbol{v}_r = \tilde{\lambda}_r\, \boldsymbol{v}_r \tag{19}$$

and note the crucial fact that $\boldsymbol{\Phi}^\intercal \boldsymbol{\Phi}$ is an $N \times N$ Gram matrix whose entries are given by

$$\big[\boldsymbol{\Phi}^\intercal \boldsymbol{\Phi}\big]_{ij} = \boldsymbol{\varphi}_i^\intercal \boldsymbol{\varphi}_j \tag{20}$$

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

$$\left[\Phi^\intercal \Phi\right]_{ij} = k_n\left(s_i, s_j\right) \tag{21}$$

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 $\boldsymbol{K} \in \mathbb{R}^{N\times N}$ with entries

$$\left[\boldsymbol{K}\right]_{ij} = k_n\left(s_i, s_j\right) \tag{22}$$

and henceforth consider the eigenvector / eigenvalue problems

$$\boldsymbol{K}\,\boldsymbol{v}_r = \tilde{\lambda}_r\,\boldsymbol{v}_r \tag{23}$$

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 $\boldsymbol{v}_r \in \mathbb{R}^N$ which, according to (14), conceptually determine the sought after feature space eigenvectors $\boldsymbol{u}_r \in \mathbb{H}$. However, we still must address several technical details.

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

To infer the correct normalization, we left-multiply (14) by $\boldsymbol{u}_r^\intercal$ to obtain $\boldsymbol{u}_r^\intercal \Phi\,\boldsymbol{v}_r = \tilde{\lambda}_r\,\boldsymbol{u}_r^\intercal \boldsymbol{u}_r$. Using (16), this can also be written as $\boldsymbol{v}_r^\intercal \boldsymbol{v}_r = \tilde{\lambda}_r\,\boldsymbol{u}_r^\intercal \boldsymbol{u}_r$. Since we want the $\boldsymbol{u}_r$ to be unit vectors, we posit $\boldsymbol{u}_r^\intercal \boldsymbol{u}_r = 1$. But this is then to say that

$$\frac{1}{\tilde{\lambda}_r}\,\boldsymbol{v}_r^\intercal \boldsymbol{v}_r = 1 \tag{24}$$

from which we deduce the required re-normalization, namely

$$\boldsymbol{v}_r \leftarrow \frac{\boldsymbol{v}_r}{\sqrt{\tilde{\lambda}_r}} \tag{25}$$

Note: It may happen that some of the eigenvalues of the kernel matrix $\boldsymbol{K}$ are 0. In other words, if we assume that the eigenvalues are ordered descendingly $\tilde{\lambda}_1 \geq \tilde{\lambda}_2 \geq \ldots \geq \tilde{\lambda}_{N-1} \geq \tilde{\lambda}_N$, there may be an $\tilde{\lambda}_p > 0$ such that $\tilde{\lambda}_{p+1} = \ldots = \tilde{\lambda}_N = 0$. 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 Φ was centered. While this will rarely be the case, a matrix which is guaranteed to be centered is $\Phi\,\boldsymbol{J}$ where

$$\boldsymbol{J} = \boldsymbol{I} - \tfrac{1}{N}\boldsymbol{1}\boldsymbol{1}^\intercal \tag{26}$$

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

$$\boldsymbol{J}^\intercal\,\Phi^\intercal \Phi\,\boldsymbol{J}\,\boldsymbol{v}_r = \tilde{\lambda}_r\,\boldsymbol{v}_r \tag{27}$$

$$\Leftrightarrow \quad \boldsymbol{J}\,\boldsymbol{K}\,\boldsymbol{J}\,\boldsymbol{v}_r = \tilde{\lambda}_r\,\boldsymbol{v}_r \tag{28}$$

$$\Leftrightarrow \quad \boldsymbol{K}_c\,\boldsymbol{v}_r = \tilde{\lambda}_r\,\boldsymbol{v}_r \tag{29}$$

Listing 4: centering a kernel matrix

```python
def center_kernel_matrix(matK):
    _, n = matK.shape
    rsum = np.sum(matK,axis=1).reshape(1,n)
    csum = rsum.reshape(n,1)
    tsum = np.sum(rsum)
    return matK - rsum/n - csum/n + tsum/n**2
```

But this is to say that we do not need to worry about centering the possibly unknown or unobtainable feature space data matrix Φ. All we need to do is to center our kernel matrix $\boldsymbol{K}$. This can be accomplished as follows

$$\boldsymbol{K}_c = \boldsymbol{J}\,\boldsymbol{K}\,\boldsymbol{J} \tag{30}$$

$$= \left[\boldsymbol{I} - \tfrac{1}{N}\,\boldsymbol{1}\boldsymbol{1}^\intercal\right]\boldsymbol{K}\left[\boldsymbol{I} - \tfrac{1}{N}\,\boldsymbol{1}\boldsymbol{1}^\intercal\right] \tag{31}$$

$$= \left[\boldsymbol{K} - \tfrac{1}{N}\,\boldsymbol{1}\boldsymbol{1}^\intercal\boldsymbol{K}\right]\left[\boldsymbol{I} - \tfrac{1}{N}\,\boldsymbol{1}\boldsymbol{1}^\intercal\right] \tag{32}$$

$$= \boldsymbol{K} - \tfrac{1}{N}\,\boldsymbol{1}\boldsymbol{1}^\intercal\boldsymbol{K} - \tfrac{1}{N}\,\boldsymbol{K}\boldsymbol{1}\boldsymbol{1}^\intercal + \tfrac{1}{N^2}\,\boldsymbol{1}\boldsymbol{1}^\intercal\boldsymbol{K}\,\boldsymbol{1}\boldsymbol{1}^\intercal \tag{33}$$

and function `center_kernel_matrix` in Listing 4 implements this in an efficient, very *numpythonic* manner.

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 $\boldsymbol{v}_r$ of our (centered) intersection string kernel matrix $\boldsymbol{K}_c$? 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 $\boldsymbol{\varphi}(s) \in \mathbb{H}$ onto feature space eigenvectors $\boldsymbol{u}_r \in \mathbb{H}$ to obtain numbers $x_r(s) \in \mathbb{R}$. In other words, the idea is to compute

$$x_r(s) = \boldsymbol{\varphi}(s)^\intercal\,\boldsymbol{u}_r \tag{34}$$

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

$$\boldsymbol{x}(s) = \begin{bmatrix} x_1(s) \\ x_2(s) \\ \vdots \\ x_d(s) \end{bmatrix} \tag{35}$$

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 $\boldsymbol{\varphi}(s), \boldsymbol{u}_r \in \mathbb{H}$.

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

$$\boldsymbol{\varphi}(s)^\intercal\,\boldsymbol{u}_r = \tfrac{1}{\tilde{\lambda}_r}\,\boldsymbol{\varphi}(s)^\intercal\,\Phi\,\boldsymbol{v}_r \tag{36}$$

Note: Division by $\tilde{\lambda}_r$ 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 its value but will leave its sign intact). We therefore simply drop this scaling and consider

$$\boldsymbol{\varphi}(s)^\intercal \boldsymbol{u}_r = \boldsymbol{\varphi}(s)^\intercal \boldsymbol{\Phi}\, \boldsymbol{v}_r \tag{37}$$

Now, looking at the expression $\boldsymbol{\varphi}(s)^\intercal \boldsymbol{\Phi}$, we realize that it involves inner products between the feature space vector $\boldsymbol{\varphi}(s)$ and the feature space vectors $\boldsymbol{\varphi}_1, \ldots, \boldsymbol{\varphi}_N$ gathered in matrix $\boldsymbol{\Phi}$. But this is to say that we may write

$$\boldsymbol{\varphi}(s)^\intercal \boldsymbol{\Phi}\, \boldsymbol{v}_r = \boldsymbol{k}(s)^\intercal \boldsymbol{v}_r = \boldsymbol{v}_r^\intercal \boldsymbol{k}(s) \tag{38}$$

where the entries of the kernel vector $\boldsymbol{k}(s) \in \mathbb{R}^N$ are given by

$$\big[\boldsymbol{k}(s)\big]_j = \boldsymbol{\varphi}(s)^\intercal \boldsymbol{\varphi}_j = k_n\big(s, s_j\big) \tag{39}$$

In other words and in conclusion, an embedding of a string s into a vector space $\mathbb{R}^d$ can be computed as

$$\boldsymbol{x}(s) = \boldsymbol{V}_d^\intercal \boldsymbol{k}(s) \tag{40}$$

where matrix

$$\boldsymbol{V}_d = \big[\boldsymbol{v}_1 \cdots \boldsymbol{v}_d\big] \in \mathbb{R}^{N \times d} \tag{41}$$

contains the (properly re-normalized) k leading eigenvectors of the centered kernel matrix $\boldsymbol{K}_c$

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

A matrix $\boldsymbol{X}$ whose columns represents embeddings of all the words in our vocabulary can thus be computed as easily as

$$\boldsymbol{X} = \boldsymbol{V}_d^\intercal \boldsymbol{K} \tag{42}$$

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

```
VOC = ["bouvier patty", ..., "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

```
n = 3
vocHists = [n_gram_hist(word, n) for word in VOC]
```

Based on this list, we can compute a corresponding intersection string kernel matrix $\boldsymbol{K}$. For this, we call `compute_kernel_matrix` in Listing 5 as follows

```
matK = compute_kernel_matrix(vocHists)
```

and obtain a corresponding *NumPy* array.

Next, we need to produce a centered kernel matrix $\boldsymbol{K}_c$. This can be accomplished via

```
matKc = center_kernel_matrix(matK)
```

Having thus computed an array representation of matrix $\boldsymbol{K}_c$, we can solve the eigenvector / eigenvalue problems $\boldsymbol{K}_c \boldsymbol{v}_r = \tilde{\lambda}_r\, \boldsymbol{v}_r$. This can happen simultaneously, if we solve the repsective matrix problem $\boldsymbol{K}_c \boldsymbol{V} = \boldsymbol{\Lambda} \boldsymbol{V}$. Using *NumPy*, this is easy. Noting that $\boldsymbol{K}_c$ is a symmetric matrix, i.e. $\boldsymbol{K}_c = \boldsymbol{K}_c^\intercal$, we should resort to the efficient function `la.eigh` which we used before [1]. Calling

```
vecL, matV = la.eigh(matKc)
```

Listing 5: computing an intersection string kernel matrix

```
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:], i):
            matK[i,j] = sum((h_i & h_j).values())
            matK[j,i] = matK[i,j]

    return matK
```

Listing 6: computing an intersection string kernel vector

```
def compute_kernel_vector(h, hs):
    vecK = np.zeros(len(hs))

    for i, h_i in enumerate(hs):
        vecK[i] = sum((h & h_i).values())

    return vecK
```

we obtain a vector $\tilde{\boldsymbol{\lambda}}$ of *ascending* eigenvalues and a matrix $\boldsymbol{V}$ whose columns contain the corresponding eigenvectors $\boldsymbol{v}_r$. Zero eigenvalues and corresponding eigenvectors can be discarded via

```
mask = vecL > 0
vecL = vecL[  mask]
matV = matV[:,mask]
```

so that the eigenvector normalization according to (25) can safely be computed as

```
matV = matV / np.sqrt(vecL)
```

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 $\boldsymbol{X} = \boldsymbol{V}_2^\intercal \boldsymbol{K}$. Here, we need to remember that `la.eigh` returns eigenvalues / eigenvectors in ascending order. Since we are interested in leading eigenvectors, we may therefore proceed as follows

```
dims = (-1,-2)
matX = matV[:,dims].T @ matK
```

If we plot the data in the columns of array `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 $\mathbb{R}^2$.

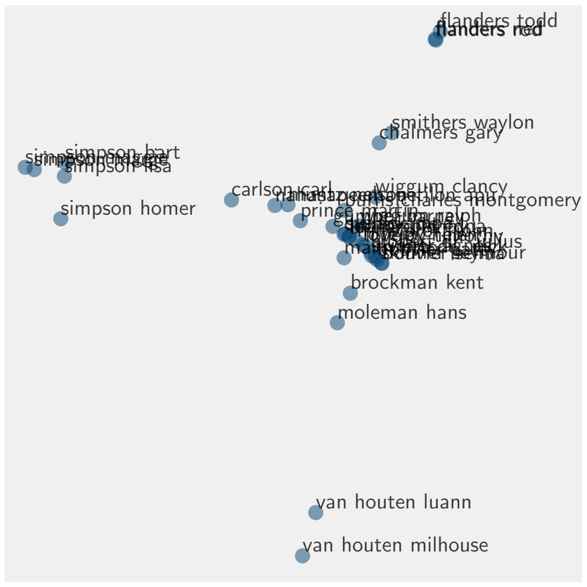
This was straightforward, wasn't it? But we also claimed that our approach extends to out-of-vocabulary words. So, to convince ourselves that it really does, we observe that our vocabulary in Fig. 1 lacks many words one would expect to see in the domain it has been sampled from. Let us therefore consider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

```
OOV = ["flanders maude", "simpson abe",
       "van houten kirk"]
```

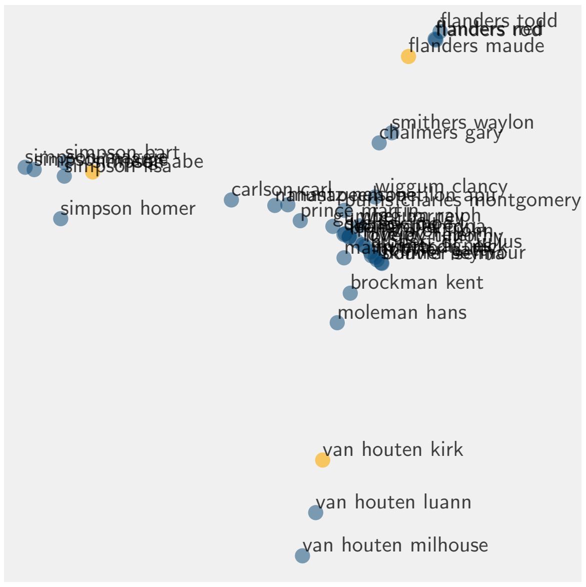
and compute kernel vectors for its elements.

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

```
hist = n_gram_hist(word, n)
vecK = compute_kernel_vector(hist, vocHists)
```



(a) 2D embedding of the vocabulary in Fig. 1



(b) additional embeddings of 3 out-of-vocabulary words

Figure 2: Examples of kPCA-based vector space embeddings of in- and out-of-vocabulary words.

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))

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 $s \in \mathcal{A}^*$ to vectors $\boldsymbol{x}(s) \in \mathbb{R}^d$. The resulting vector space embeddings reflect syntactic commonalities among given words and can thus provide useful structural information for a variety of downstream task.

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

[1] C. Bauckhage. 2015. NumPy / SciPy Recipes for Data Science: Eigenvalues / Eigenvectors of Covariance Matrices. researchgate.net. https://dx.doi.org/10.13140/RG.2.1.2307.5046.
[2] C. Bauckhage. 2022. *ML2R Coding Nuggets: Intersection String Kernels for Language Processing.* Technical Report. MLAI, University of Bonn.
[3] C. Bauckhage and P. Welke. 2021. *ML2R Theory Nuggets: Centering Data- and Kernel Matrices.* Technical Report. MLAI, University of Bonn.

[4] M. Beeksma, M. van Gompel, F. Kunneman, L. Onrust, B. Regnerus, D. Vinke, E. Brito, C. Bauckhage, and R. Sifa. 2018. Detecting and Correcting Spelling Errors in High-quality Dutch Wikipedia Text. *Computational Linguistics in the Netherlands J.* 8 (2018), 122–137.

[5] D. Biesner, R. Ramamurthy, M. Lübbering, B. Fürst, H. Ismail, L. Hillebrand, A. Ladi, M. Pielka, R. Stenzel, T. Khameneh, V. Krapp, I. Huseynov, J. Schlums, U. Stoll, U. Warning, B. Kliem, C. Bauckhage, and R. Sifa. 2020. Leveraging Contextual Text Representations for Anonymizing German Financial Documents. In *Proc. Knowledge Discovery from Unstructured Data in Financial Services*. AAAI.

[6] D. Biesner, R. Ramamurthy, R. Stenzel, M. Lübbering, L. Hillebrand, A. Ladi, M. Pielka, R. Loitz, C. Bauckhage, and R. Sifa. 2022. Anonymization of German Financial Documents Using Neural Network-based Language Models with Contextual Word Representations. *Int. J. of Data Science and Analytics* 13, 2 (2022).

[7] E. Brito, B. Georgiev, D. Domingo-Fernandez, C.T. Hoyt, and C. Bauckhage. 2019. RatVec: A General Approach for Low-dimensional Distributed Vector Representations via Rational Kernels. In *Proc. KDML-LWDA*.

[8] E. Brito, R. Sifa, and C. Bauckhage. 2017. KPCA Embeddings: An Unsupervised Approach to Learn Vector Representations of Finite Domain Sequences. In *Proc. KDML-LWDA*.

[9] E. Brito, R. Sifa, C. Bauckhage, R. Loitz, U. Lohmeier, and C. Pünt. 2019. A Hybrid AI Tool to Extract Key Performance Indicators from Financial Reports for Benchmarking. In *Proc. Symposium on Document Engineering*. ACM.

[10] T. Brown and et al. 2020. Language Models Are Few-Shot Learners. In *Proc. NeurIPS*.

[11] C.L. Chapman, L.P. Hillebrand, M.R. Stenzel, T. Deusser, D. Biesner, C. Bauckhage, and R. Sifa. 2022. Towards Generating Financial Reports from Tabular Data Using Transformers. In *Proc. Cross Domain Conf. for Machine Learning and Knowledge Extraction (CD-MAKE)*.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proc. NAACL*.

[13] V. Gupta, S. Giesselbach, S. Rüping, and C. Bauckhage. 2019. Improving Word Embeddings Using Kernel PCA. In *Proc. Workshop on Representation Learning for NLP @ ACL*.

[14] L.P. Hillebrand, D. Biesner, C. Bauckhage, and R. Sifa. 2020. Interpretable Topic Extraction and Word Embedding Learning Using Row-Stochastic DEDICOM. In *Proc. Cross Domain Conf. for Machine Learning and Knowledge Extraction (CD-MAKE)*.

[15] L.P. Hillebrand, T. Deußer, T. Dilmaghani Khameneh, B. Kliem, R. Loitz, C. Bauckhage, and R. Sifa. 2022. KPI-BERT: A Joint Named Entity Recognition and Relation Extraction Model for Financial Reports. *arXiv:2208.02140 [cs.CL]* (2022).

[16] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proc. NIPS*.

[17] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).

[18] M.E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. 2018. Deep Contextualized Word Representations. In *Proc. NAACL*.

[19] M. Pielka, A. Ladi, C. Chapman, E. Brito, R. Ramamurthy, P. Mayer, A. Wahab, R. Sifa, and C. Bauckhage. 2020. Using Ensemble Methods and Sequence Tagging to Detect Causality in Financial Documents. In *Proc. FinCausal*.

[20] M. Pielka, R. Sifa, L. Hillebrand, D. Biesner, R. Ramammurthy, A. Ladi, and C. Bauckhage. 2021. Tackling Contradiction Detection in German Using Machine Translation and End-to-End Recurrent Neural Networks. In *Proc. ICPR*.

[21] R. Ramamurthy, M. Pielka, R. Stenzel, C. Bauckhage, R. Sifa, T. Khameneh, U. Warning, B. Kliem, and R. Loitz. 2021. ALiBERT: Improved Automated List Inspection (ALI) with BERT. In *Proc. Symposium on Document Engineering*. ACM.