ABSTRACT
This is the first in a miniseries of notes on kernel methods for
language processing. We discuss the idea of measuring \( n \)-gram similarities of words by computing intersection string kernels and
demonstrate that the Python standard library allows for compact
implementations of this idea.

1 INTRODUCTION
Most machine learning methods for natural language processing
require numerical vector representations of words. Nowadays, such
word embeddings are usually computed using context-free or bi-
directional contextual models. These involve architectures such
as skip gram- or transformer networks and can learn semantic
representations [9, 11, 17–19].

Yet, simpler syntax-based representations still have their merits,
too. They offer benefits when processing texts of morphologically
rich languages such as German or Turkish [5], cope with infrequent
or out-of-vocabulary words [3, 12], and can be trained on small
corpora [7]. Last but not least, the underlying learning algorithms
are easy to implement [6].

This note practically demonstrates the latter. In particular, we
show that plain vanilla Python makes it super easy to implement
code that computes intersection string kernels [14].

String kernel such as these have many practical applications. For
instance, in combination with kernel PCA, they allow for embed-
ding words into the space spanned by the feature space principal
components of a given vocabulary. This can be used to visualize
whole vocabularies (see Fig. 1) and seamlessly extends to out-of-
vocabulary words.

Yet, in order to keep this note short, we leave the discussion of
theory and practice of such applications to later.

The main part of this note is section 2 where we look at theory
and practice alike. Indeed, for experienced Python programmers, it
may be downright trivial to implement the techniques we discuss.
Yet, as the underlying theory is not necessarily trivial, it seems
appropriate to interlace our theoretical discussion with practical
coding examples.

Another major part is found in the appendix where we show that
intersection string kernels are actually Mercer kernels. However,
the material presented there is not essential to readers who are
mainly interested in the more practical aspects of the topic covered
in this note.

As always, readers are expected to be passingly familiar with
Python. Those who would like to experiment with the simple code
snippets we provide only need to

```python
from collections import Counter
```

Figure 1: A didactic vocabulary of words and 2D embeddings
computed via KPCA on intersection string kernel matrices.
2 THEORY & PRACTICE

Given the venerable history of string kernels [15, 16], intersection string kernels [14] appeared surprisingly late in the game. The surprise is that they are essentially but histogram intersection kernels which have long been used in computer vision [1].

In this section, we discuss the underlying ideas and basic ingredients of intersection string kernels. Since Python makes it very easy to implement them, practical coding examples will be interlaced with our theoretical discussion. First, however, we need to recall several more basic concepts.

An alphabet $\mathcal{A}$ of size $|\mathcal{A}| = m$ is a set of symbols $\{a_1, \ldots, a_m\}$. Since our overriding interest lies in language processing, all our examples in this note will consider the alphabet

$$\mathcal{A} = \{\text{a}, \text{b}, \ldots, \text{z}, \ldots\}$$

(1)
of lower case Latin letters together with the character _ which denotes a blank space.

A string $s$ over an alphabet $\mathcal{A}$ is a sequence of symbols $s \in \mathcal{A}^*$ where $\mathcal{A}^*$ is the Kleene star or set of all possible sequences (finite or infinite) of symbols in $\mathcal{A}$. If a string $s$ is a sequence of $l$ symbols, it is a string of length $|s| = l$. It is also an element of $\mathcal{A}^l \subset \mathcal{A}^*$ which is the set of all strings of length $l$ over $\mathcal{A}$. If the size of $\mathcal{A}$ is $m$, then the size of $\mathcal{A}^*$ is $m^l$.

2.1 $n$-Grams

The multiset of $n$-grams (or $n$-spectrum) of a string $s$ is the multiset $\mathcal{M}_n(s)$ of all its contiguous sub-strings of length $n$.

Since the meaning behind this definition becomes immediately apparent from looking at examples, let us consider these exemplary strings over the alphabet in (1)

$$s_1 = "\text{homer simpson}"$$

$$s_2 = "\text{lenny leonard}"$$

$$s_3 = "\text{ned flanders}"$$

Their multisets of bi-grams ($n = 2$) are

$$\mathcal{M}_2(s_1) = \{"\text{ho}", "\text{om}", "\text{me}", "\text{er}", "r \ " , "s", "si", "im", "mp", "ps", "so", "on"\}$$

$$\mathcal{M}_2(s_2) = \{"\text{le}", "\text{en}", "\text{nn}", "\text{ny}", "y \ " , "1", "le", "eo", "on", "na", "ar", "rd"\}$$

$$\mathcal{M}_2(s_3) = \{"\text{ne}", "\text{ed}", "\text{d \ " , "f", "f1", "1a", "an", "nd", "de", "er", "rs"\}$$

and their multisets of tri-grams ($n = 3$) are

$$\mathcal{M}_3(s_1) = \{"\text{hom}", "\text{ome}", "\text{mer}", "\text{er} \ " , "r \ s", " si", "sim", "imp", "mps", "psn", "son"\}$$

$$\mathcal{M}_3(s_2) = \{"\text{len}", "\text{enn}", "\text{nnyn}", "\text{ny} \ " , "1", "le", "leo", "eon", "ona", "nar", "ard"\}$$

$$\mathcal{M}_3(s_3) = \{"\text{ned}", "\text{ed} \ " , "d \ f", " f1", "fla", "lan", " and", "nde", "der", "ers"\}$$

Just to be clear, we reiterate that our examples in this note treat blank spaces as normal charters. That is, we understand strings such as “homer simpson” as a single word which happens to contain a blank. Such blanks therefore occur in several of the above bi- and tri-grams.

We also point out that a multiset generalizes the notion of a set. Whereas the elements $x$ of a set $\mathcal{X}$ must be distinct, i.e. only occur once, a multiset $\mathcal{Y}$ allows for multiple instances for each of its elements $y$. The number of occurrences of an element $y \in \mathcal{Y}$ is called its multiplicity and is denoted by $\mathcal{m}_\mathcal{Y}(y)$. For instance, for the bi-grams “le” and “en” in the above multiset $\mathcal{M}_2(s_2)$, we have

$$\mathcal{m}_{\mathcal{M}_2(s_2)}("\text{le}") = 2$$

$$\mathcal{m}_{\mathcal{M}_2(s_2)}("\text{en}") = 1$$

In Python, the natural data structure for multisets are lists. Even better, Python makes it ridiculously easy to compute $n$-grams of text strings. This is because it treats strings as iterable objects and provides a str class for convenient string processing. Our function n_grams in Listing 1 takes full advantage of these features and was previously explained in [2]. Using it, we can create $n$-gram lists for arbitrary strings. For instance

```python
for n in [2,3,4]:
    print (list(n_grams("homer simpson", n)))
```

results in

```python
>>> ['ho', 'om', 'me', 'er', 'r ', ' s', 'si', 'im', 'mp', 'ps', 'so', 'on']
>>> ['hom', 'ome', 'mer', 'er', 'r s', ' si', 'sim', 'imp', 'mps', 'psn', 'son']
>>> ['home', 'omer', 'mer ', 'er s', 'r si', ' sim', 'simp', 'imps', 'mpso', 'pson']
```

which agrees with what we would expect from our discussion up to this point.

Now that we know how to compute $n$-grams, let us have a closer look at, say, the bi-grams of our exemplary strings $s_1$, $s_2$, and $s_3$. In particular, let us see if there are any commonalities.

Apparently, $s_1$ and $s_2$ share the bi-gram “on”. Moreover, $s_1$ and $s_3$ share the bi-gram “er” whereas $s_2$ and $s_3$ do not share any bi-gram at all.

Based on observations like these, it seems reasonable to say that strings $s_1$ and $s_2$ as well as $s_1$ and $s_3$ are somewhat similar. Strings $s_2$ and $s_3$, on the other hand, appear to be dissimilar.

In general, we could of course also resort to $n$-grams with $n > 2$ to come up with such statements about similarity. But how could or rather should we quantify the $n$-gram similarity of two any strings $s_i$ and $s_j$ ?

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

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

1 Usually, we would treat strings with blank spaces as sequences of two or more words, say “homer” and “simpson”. But, for this note, we deliberately decided not to do this.
While this looks horrible, it simply means that \( \in A \) over \( s \) which results in

Again, this looks horrible but is nothing but a formal way of saying

There are (at least) two ways of formalizing the respective counting

or as a relation

Objects. Its use is shown in function

is easily implemented, too. This is because the

is the set (not multiset!) of the n-grams \( g \in A^n \) that occur in \( s \).

For any string \( s \in A \), we can determine \( M_n(s) \) and \( S_n(s) \). Given these, we can define the n-gram histogram \( h_{n,s} \) of \( s \) as a relation or set of pairs

Again, this looks horrible but is nothing but a formal way of saying that our histogram pairs every n-gram in \( s \) with the number of times it occurs in \( s \).

Working with Python, this view on n-gram histograms of strings is easily implemented, too. This is because the collections module provides the dictionary subclass Counter for counting hashable objects. Its use is shown in function n_gram_hist in Listing 2 and its effect is once again best understood by looking at examples.

For instance, the bi-gram histograms of two of our exemplary strings can be produced and printed using

Looking at these exemplary outputs, we observe that Python counters store objects to be counted as dictionary keys and their counts as dictionary values. Our exemplary outputs further indicate that n_gram_hist works as intended.

### 2.2 n-Gram Histograms

Our first step towards quantifying n-gram string similarities is to compute n-gram histograms of strings. As we shall see, the formal specification of this idea is more involved than its practical implementation. But let us be formal anyway.

To begin with, we recall that the purpose of a histogram is to count how often certain objects appear in some collection. In our case, the objects to be counted are n-grams and the collection they are to be counted in is the multiset of n-grams of a given string.

There are (at least) two ways of formalizing the respective counting mechanism. We can think of the n-gram histogram \( h_{n,s} \) of a string \( s \) over \( A \) as a discrete function

or as a relation

Both points of view have their merits for our purposes in this note, but, for now, we will only consider the second one.

This requires us to recall the notion of the support of a multiset which is but the underlying set of a multiset. In our context, the support \( S_n(s) \) of the multiset \( M_n(s) \) of n-grams of a string \( s \in A \) is given by

While this looks horrible, it simply means that \( S_n(s) \) is the set (not multiset!) of the n-grams \( g \in A^n \) that occur in \( s \).

For any string \( s \in A \), we can determine \( M_n(s) \) and \( S_n(s) \). Given these, we can define the n-gram histogram \( h_{n,s} \) of \( s \) as a relation or set of pairs

Again, this looks horrible but is nothing but a formal way of saying that our histogram pairs every n-gram in \( s \) with the number of times it occurs in \( s \).

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

```python
def n_gram_hist(s, n):
    return Counter(n_gram(s, n))
```

2.3 Intersection String Kernels

Now that we can practically compute n-gram histograms, we can take our second and final step towards quantifying the n-gram similarity of two strings \( s_i \) and \( s_j \). Again, practice will be easier than theory.

To begin with, we introduce the shorthand

\[
    h_{n,s}(g) = m_{M_n(s)}(g)
\]

(6)
to denote the count of n-gram \( g \) in string \( s \). Next, we let the set

\[
    I_n = S_n(s_i) \cap S_n(s_j)
\]

(7)
denote the intersection of the support sets of n-grams of \( s_i \) and \( s_j \).

With these definitions at hand, we then compute the function

\[
    k_n(s_i, s_j) = \sum_{g \in I_n} \min \{ h_{n,s_i}(g), h_{n,s_j}(g) \}
\]

(8)
which counts how many n-grams the two strings have in common.

To better understand the rationale behind the function or string similarity measure in (8), we consider another example involving simple bi-gram histograms and the following two strings

\[
    s_i = "\text{lenny leonard}"
\]

\[
    s_j = "\text{hans moleman}"
\]

Both these strings contain the bi-gram \( g = "1e" \). In fact, this is the only bi-gram they share so that \( I_2 = \{ "1e" \} \). In \( s_i \), "1e" occurs twice but in \( s_j \) it only occurs once. Hence, \( h_{n,s_i}(g) = 2 \), \( h_{n,s_j}(g) = 1 \), and \( \min \{2, 1\} = 1 \). In this example, the bi-gram similarity of strings \( s_i \) and \( s_j \) thus amounts to 1.

(For those who are interested in even more jargon: What we are computing in (8) is nothing but the size of the multiset intersection of \( M_n(s_i) \) and \( M_n(s_j) \). But let us leave it at that . . . )

Working with Python, the computation of (8) is a again a breeze. This is because the operator & allows for intersecting counters. That is, we neither have to worry about computing \( I_n \) nor about computing minima. For instance, to replicate parts of the example we just went through, we may use

```python
hi = n_gram_hist("lenny leonard", 2)
jh = n_gram_hist("hans moleman", 2)
print (hi & jh)
```

This results in

```python
>>> Counter(('1e': 1))
```

which tells us that bi-gram "1e" is shared once by our strings.

To replicate our example in full, i.e. to also sum over all the bi-grams \( g \in I_2 = S_2(s_i) \cap S_2(s_j) \), we simply use

```python
print (sum((hi & jh).values()))
```

which yields

```python
>>> 1
```
Summarizing all these computations in a single function leads to `intersection_str_kernel` as shown in Listing 3.

That is it! We now have a notion for the n-gram similarity of two strings and may use it in language processing applications.

However, for curious readers, there may remain a lingering open question, namely:

Q: Why have we been talking about intersection string kernels?
Or, to paraphrase a bit more technically:

Q: Why is our string similarity function in (8) called $k_n(\cdot, \cdot)$?
Well, to be brief, this is because of the very crucial fact that

A: Our string similarity function $k_n(\cdot, \cdot)$ is a Mercer kernel!

Proving this momentous claim is not that difficult but somewhat tedious. We therefore defer this to the appendix but encourage our readers to go through the arguments presented there. For now, we will look at a very basic practical application of what we just worked out.

2.4 Intersection String Kernel Matrices

In order to provide a first glimpse at what to practically do with intersection string kernels, we next compute $n \in \{2, 5\}$-gram similarity matrices for the 36 words in the vocabulary in Fig. 1(a).

To this end, we first of all assume that they are given in form of a list of strings

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

Using this list, we then compute a list of n-gram histograms. Opting for $n = 2$, this can be accomplished by means of

\[ \text{ngramHist} = [\text{n}\_\text{gram}\_\text{hist}(	ext{word}, n) \text{ for word in VOC}] \]

Using this list, we then compute a similarity matrix $S \in \mathbb{R}^{36 \times 36}$.

Note: The following way of implementing matrix $S$ as a list of lists $5$ is of course very bad practice! However, in this note, we deliberately opted not to use any NumPy functionalities (just to show that basic language processing can be done without it) ...

```python
m = len(VOC)
S = [[0] * m for i in range(m)]
for i, hi in enumerate(vocHists):
    for j, hj in enumerate(vocHists[i:]):
        S[i][j] = sum((hi & hj).values())
```

If we then print the resulting lists of lists, we obtain something as shown in Fig. 2, and, if we repeat the whole exercise with $n = 5$, we obtain a result as in Fig. 3.

Note: In order to improve readability, neither figure shows the numerous 0s contained in either similarity matrix.

What is strikingly apparent is that the 5-gram similarity matrix is much sparser than the 5-gram similarity matrix. This was to be expected, because the longer an $n$-gram in a given word, the less likely it reoccurs in another word. We also observe the seven blocks or clusters along the main diagonals of both matrices. These reflect the syntactic similarities of the names of the members of the Bouvier, Flanders, Lovejoy, Simpson, Skinner, van Houten and Wiggum families.

All in all, these results are rather silly. Nevertheless, they indicate that n-gram similarities can reveal latent structures within a given vocabulary of words. More serious and more useful applications of n-gram similarities will be discussed in future notes.

```python
Listing 3: computing the intersection kernel of strings $s_i, s_j$

```
3 SUMMARY AND OUTLOOK

Machine learning for natural language processing is a cornerstone for tasks such as intelligent document analysis [4, 8, 10, 13, 20, 21]. Many methods in this arena require numerical vector representations of words which are currently most commonly computed using neural networks [9, 11, 17–19]. However, light weight kernel methods have their merits, too [3, 6, 7, 12].

In this note, we had a first look at string kernels and studied the notion of intersection string kernels. We saw that they have one truly appealing property, namely ease of implementation. Our plain vanilla Python implementations for computing syntactic, i.e. n-gram based, string similarities merely involved 8 lines of code (2 lines in Listing 1, 2 lines in Listing 2, and 4 lines in Listing 3).

Moreover, the insight that an intersection string kernel is indeed a Mercer kernel will be of practical value, too, for it allows us to unleash everything we know about kernel machines on language processing. We will further elaborate on this in upcoming notes where we will study ideas such kernel PCA for word embeddings or kernel SVMs for language disambiguation.

APPENDIX

In this appendix, we work out why and how the string similarity function \( k_n(s_i, s_j) \) defined in (8) is indeed a Mercer kernel. But we best start with a disclaimer: The constructions we present next are of purely theoretical interest. They are not meant to ever be implemented in practice. Rather, they are intended to prove that (8) has a hidden but important aspect to it.

To establish that \( k_n(s_i, s_j) \) is a Mercer kernel, we will consider n-gram histograms from the point of view we did not follow up on in the main text of this note. That is, we will think of the n-gram histogram \( h_{n,s} \) of a string \( s \) over \( A \) as a discrete function

\[
h_{n,s} : \mathcal{A}^n \rightarrow \mathbb{N}
\]  

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

We also recall that, for a finite alphabet \( \mathcal{A} \) of size \( m \), the finite size of \( \mathcal{A}^n \) is \( m^n \). This finiteness allows for a unique mapping between the \( g \in \mathcal{A}^n \) and the numbers \( r \in \mathbb{R} = (1, 2, \ldots, m^n) \in \mathbb{N} \). For instance, for the set of tri-grams \( \mathcal{A}^3 \) over the alphabet in (1), we may consider the canonical assignment

\[
\begin{align*}
\text{aaa} & \leftrightarrow 1 \\
\text{aab} & \leftrightarrow 2 \\
\text{aac} & \leftrightarrow 3 \\
\vdots 
\end{align*}
\]

We may then write \( g[i] \) to refer to the \( i \)-th n-gram in \( \mathcal{A}^n \). This allows us to reconsider the n-gram histogram \( h_{n,s} \) as a function

\[
h_{n,s} : \mathcal{R} \rightarrow \mathbb{N}
\]

where \( h_{n,s}[r] = m_{M_n(s)}(g[r]) \) and, by convention, \( h_{n,s}[r] = 0 \) if \( g[r] \) is not a substring of \( s \).

Even more, since the domain \( \mathcal{R} \) of this function is finite, we may identify \( h_{n,s} \) with a vector \( \mathbf{h}(s) \in \mathbb{R}^{m^n} \) where

\[
\mathbf{h}(s) = \left[ \begin{array}{c} h_{n,s}[1] \\ h_{n,s}[2] \\ \vdots \\ h_{n,s}[m^n] \end{array} \right]
\]

Note that vector \( \mathbf{h}(s) \) does not need to carry a subscript \( n \) because this piece of information is implicitly encode in its dimensionality. Further note that we henceforth write

\[
[\mathbf{v}]_r
\]

to refer to the \( r \)-th entry of a vector \( \mathbf{v} \).

Intersection String Kernels Are Mercer Kernels

Given what we just worked out, we observe that function \( k_n(s_i, s_j) \) in (8) can just as well written as

\[
k_n(s_i, s_j) = \sum_{r=1}^{m^n} \min \left\{ [\mathbf{h}_1]_r, [\mathbf{h}_j]_r \right\}
\]

where

\[
\mathbf{h}_i = \mathbf{h}(s_i) \tag{13}
\]
\[
\mathbf{h}_j = \mathbf{h}(s_j) \tag{14}
\]

Now, if \( k_n(s_i, s_j) \) was a Mercer kernel, it must also be an inner product in some latent, i.e. typically unknown, Hilbert space \( \mathcal{H} \). In other words, if \( k_n(s_i, s_j) \) was a Mercer kernel, we must be able to equivalently compute it as

\[
k_n(s_i, s_j) = \mathbf{\varphi}_i^\top \mathbf{\varphi}_j \tag{15}
\]

where

\[
\mathbf{\varphi}_i = \mathbf{\varphi}(s_i) \tag{16}
\]
\[
\mathbf{\varphi}_j = \mathbf{\varphi}(s_j) \tag{17}
\]

and \( \mathbf{\varphi} : \mathcal{A}^* \rightarrow \mathcal{H} \) is an appropriate feature map that takes strings in \( \mathcal{A}^* \) to vectors in \( \mathcal{H} \).

Our problem at this point is thus find such a feature map. That is, to prove that \( k_n(s_i, s_j) \) is a Mercer kernel, we need prove that there exists at least one function \( \mathbf{\varphi} : \mathcal{A}^* \rightarrow \mathcal{H} \) such that (12) can equivalently be written as (15).

Looking at the systems of equations in (12)–(14) and (15)–(17), their structures appear encouragingly similar. That is, it does not seem impossible to establish a connection between them.

The main difficulty is that the right hand side of (12) involves non-linear functions (min) and therefore does not constitute an inner product. (If it did, our job was already done because we could simply let \( \mathbf{\varphi}(s) = \mathbf{h}(s) \).

However, the entries \( [\mathbf{h}(s)]_r \) of the histogram vector \( \mathbf{h}(s) \) are special in that they are counting numbers. And counting numbers can be represented using the following (not really practical but
that is, we can represent each \( h(s) \), of \( h(s) \) either as a zero or as a sequence of ones. To let all such sequences be of the same length, we may right-pad them with an appropriate amount of 0s. To this end, we note that a string of length \( l \) contains \( l - n + 1 \) contiguous substrings or \( n \)-grams. Hence, we can identify each \([h(s)]_r\) of a string \( s \) with a binary vector

\[
z_r(s) = \left[ \begin{array}{l} 1 \ 1 \ \ldots \ 1 \ 0 \ 0 \ \ldots \ 0 \\
\end{array} \right]_{[h(s)]_r}
\]

because

\[
[h(s)]_r = \sum_{q=1}^{l} [z_r(s)]_q
\]

However, if we do this for two strings \( s_i \) and \( s_j \), their lengths \( l_i \) and \( l_j \) may differ. A remedy is to consider a maximum length \( L \) longer than the length of any string we will ever encounter. For the point we are making here, the exact choice of \( L \) does not matter, it is really but a theoretical construct. When in doubt, we could chose it to be a ridiculously large number such as, say, \( L = 10^{60} \), the number of atoms in the known universe. Assuming an appropriate \( L \), we can represent each \([h(s)]_r\), as

\[
z_r(s) = \left[ \begin{array}{l} 1 \ 1 \ \ldots \ 1 \ 0 \ 0 \ \ldots \ 0 \\
\end{array} \right]_{[h(s)]_r}
\]

Next, we observe the following very peculiar property of binary numbers \( x, y \in \{0, 1\} \), namely

\[
\min\{x, y\} = x \cdot y
\]

Just because of this special property of binaries, we can now actually write

\[
\min\{h(s_i)\}_r, h(s_j)\}_r = \min\left\{ \sum_{q=1}^{L} [z_r(s_i)]_q \right\} \sum_{q=1}^{L} [z_r(s_j)]_q
\]

\[
= \sum_{q=1}^{L} \min\left\{ [z_r(s_i)]_q \right\} [z_r(s_j)]_q
\]

\[
= \sum_{q=1}^{L} [z_r(s_i)]_q \cdot [z_r(s_j)]_q
\]

\[
= z_r(s_i)^T z_r(s_j)
\]
Intersection String Kernels for Language Processing


