# String Subsequence Kernels for Text Classification

**Michael Giuffrida**
**April 30, 2012**

**Yale University**
**CPSC 490: Senior Project**

**Advisor: Dana Angluin**

## Abstract

This paper explores the string subsequence kernel, a kernel function whose feature space is generated by subsequences of strings. This kernel compares two strings based on the number of occurrences of common substrings they contain, where each common substring is weighted based on how contiguous that substring is within the string. Although a recursive definition of the string subsequence kernel that uses dynamic programming methods exists, its computation is still expensive. An approximation, the string subsequence kernel with lambda pruning, is also considered, and a novel extension of the approximation algorithm that includes caching is investigated. Once the accuracies of the string subsequence kernel are presented, the runtimes for the approximation kernels are measured and compared with the exact algorithm over a variety of parameters.

# Contents

# 1 Background

The support vector machine (SVM) is a method of predicting the classification of input data. Given a set of positive and negative training examples of a class, a binary SVM classifier represents the examples in a high-dimensional space and finds the separating hyperplane between the two classes with the highest margin. If the input data is not linearly separable, it can be transformed into a higher-dimensional feature space. For efficient computation, a kernel function may be used to directly compute the inner product between data points in the feature space without explicitly representing the data in the higher-dimensional feature space, thus avoiding the high computational costs associated with feature extraction in the higher-dimensional space.

## 1.1 String Subsequence Kernel

The kernel trick can be useful in text classification, with applications in fields such as document classification, bioinformatics[2], and spam filtering[4]. For such purposes, Lodhi et al. have developed a string subsequence kernel (SSK) that compares two strings based on the number of occurrences of common substrings they contain.[3] Each common substring is weighted based on how contiguous that substring is within the string. The feature space is indexed by all possible substrings of $n$ characters, where the feature vector of a sample string consists of the weights associated with each of these features. Specifically, the value in the vector corresponding to a particular substring is zero if that substring is not a subsequence of the sample; otherwise, the value is $\lambda^l$, where $l$ is the length of the subsequence in the document and $\lambda$ is a decay factor in $(0, 1]$.

More formally, let $s$ and $t$ be strings from the finite alphabet $\Sigma$, where the length of $s = s_1 \dots s_{|s|}$ is $|s|$. The substring $s_i \dots s_j$ is denoted by $s[i:j]$. The target string $u$ is a subsequence of $s$ (denoted $u = s[\mathbf{i}]$) if there exist increasing indices $\mathbf{i} = (i_1, \dots, i_{|u|})$ such that $u_j = s_{i_j}$ for each character $u_j$ in $u$. The length $l(\mathbf{i})$ of this subsequence is given by $i_{|u|} - i_1 + 1$, which represents the number of characters that the subsequence $s[\mathbf{i}]$ spans in the string $s$.

Lodhi et al. define the feature mapping for a string by giving the u coordinate for each $\mathrm{u} \in \Sigma^{\mathrm{n}}$ as

$$\phi_u(s) = \sum_{\mathbf{i}:u=s[\mathbf{i}]} \lambda^{l(\mathbf{i})},$$

which represents the number of occurrences of the subsequence in $s$, weighted by their contiguities. The kernel function represents the inner product of the feature vectors for strings $s$ and $t$ as the sum of the products of the above $u$ coordinate for all substrings $u$:

$$K_n(s,t) = \sum_{u\in\Sigma^n} \langle \phi_u(s) \cdot \phi_u(t) \rangle = \sum_{u\in\Sigma^n} \sum_{\mathbf{i}:u=s[\mathbf{i}]} \lambda^{l(\mathbf{i})} \sum_{\mathbf{j}:u=t[\mathbf{j}]} \lambda^{l(\mathbf{j})} = \sum_{u\in\Sigma^n} \sum_{\mathbf{i}:u=s[\mathbf{i}]} \sum_{\mathbf{j}:u=t[\mathbf{j}]} \lambda^{l(\mathbf{i})+l(\mathbf{j})}$$

Thus, $n$ is the length of common substrings that contribute to the kernel. The weight decay factor $\lambda$ controls how the length of a subsequence within a string is penalized; lower values of $\lambda$ place

lower weights on less contiguous subsequences, and a value of 1 does not penalize interior gaps at all. This kernel is similar to the *n*-grams kernel, which compares strings using common substrings but does not allow gaps in the subsequences.

To compute SSK efficiently, Lodhi et al. introduce a recursive kernel:

$$K_i'(s,t) = \sum_{u \in \Sigma^i} \sum_{\mathbf{i}: u=s[\mathbf{i}]} \sum_{\mathbf{j}: u=t[\mathbf{j}]} \lambda^{|s|+|t|+i_1+j_1+2}, i = 1,..,n-1$$

so that

$$K_i'(s,t) = K_i(s,t) = 0, \qquad if \min(|s|,|t|) < i,$$

$$K_i'(sx,t) = \lambda K_i'(s,t) + \sum_{j: t_j = x} K_{i-1}'(s, t[1:j-1])\lambda^{|t|-j+2},$$

$$K_n(sx,t) = K_n(s,t) + \sum_{j: t_j = x} K_{n-1}'(s, t[1:j-1])\lambda^2$$

As Lodhi et al. show, this string subsequence kernel can be evaluated in time $O(n|s||t|)$. They explore the use of SSK on the Reuters dataset, varying the length of the subsequence $n$ as well as the decay factor $\lambda$. The $F_1$ score of SSK is close to, but rarely better than, that of the *n*-grams kernel for this dataset, although both performed better than the standard linear word kernel. The optimal sequence length was found to be fairly small (4 to 7 characters), suggesting that smaller substrings are more useful in classification than larger substrings. Furthermore, SSK was found to perform best with the decay factor $\lambda$ at a very small value, thus highly penalizing interior gaps. While the precision peaked at much higher values of $\lambda$, this corresponded to a drop in sensitivity.

## 1.2 Lambda Pruning

Seewald and Kleedorfer present an approximation of this string kernel, SSK with lambda pruning (SSK-LP).[5] In contrast to SSK, SSK-LP with lambda pruning uses less memory and can be less computationally complex. Because $K_n(s, t)$ is a sum over powers of lambda, one can "prune" those powers of lambda that have little effect on the overall result, thereby limiting the depth of recursion during computation. SSK-LP ignores terms in which the powers to which $\lambda$ is raised are so large that the contribution of the terms is very small. That is, when the sum of lengths of a subsequence in $s$ and $t$ is greater than some threshold $\theta$, we end the recursive computation:

$$K_{n,\theta}(s,t) = \sum_{u \in \Sigma^n} \sum_{\mathbf{i}: u=s[\mathbf{i}]} \sum_{\mathbf{j}: u=t[\mathbf{j}]} f(l(\mathbf{i}), l(\mathbf{j}))$$

$$f(x,y) = \begin{cases} 0, & x+y > \theta \\ \lambda^{x+y} & \text{otherwise} \end{cases}$$

If $\theta$ is set to $2n$, the kernel will only match contiguous substrings, just like an *n*-grams kernel; for a better approximation of SSK, $\theta$ should be set to higher multiples of $n$.

The recursive definition of SSK-LP is similar to that of SSK, except that its component functions include an additional parameter, $m$, giving the number of times the recursion result has been multiplied by $\lambda$ (i.e., the lengths of the substrings being considered). This limits the depth of recursion, but it makes a cache of intermediate results impractical, as such a cache would have to include an extra dimension for the possible values of $m$. For even moderate values of $\theta$, this can result in intractably high memory consumption. Without caching, the approximation presents a tradeoff between speed and accuracy.

Support vector machines using the approximated method have been shown to achieve accuracies similar to those using the exact SSK algorithm. Although the degree of approximation of SSK-LP can be tuned using the parameter $\theta$, in practice the kernel is only efficient within a small approximation window because of its lack of caching. As shown by Seewald and Kleedorfer, for certain parameters, SSK-LP is slower when compared to SSK with caching. As $n$ increases, the highest $\theta$ at which SSK-LP is still faster than SSK decreases. This makes SSK-LP less useful for approximation of SSK for moderate to high values of $\lambda$, where powers of lambda drop off more slowly and SSK-LP is less accurate. However, it may be possible for lambda pruning to scale better with higher $\theta$ by extending SSK-LP to include caching, as shown in section 3.1. One goal of this project was to determine whether SSK-LP with caching (SSK-LPC) is efficient at higher values of $\theta$, for situations where values of $\theta$ for which SSK-LP is feasible do not result in accurate enough approximations. If so, applications in which higher values of $\lambda$ yield better results may be able to use SSK-LPC when SSK-LP and SSK are too slow.

SSK may be useful in domains other than document classification, because many problems can be reduced to string processing. For example, the spectrum kernel is an implementation of the *n*-grams kernel used to compare biological sequences by finding common substrings in strings such as protein sequences.[2] The string subsequence kernel may be particularly effective in spam classification, due to its robustness in the face of obfuscated words, but online spam filtering places limits on the computational costs of the algorithms used.[4] The possibility of learning formal languages with SSK is explored by Kontorovich et al., whose subsequence kernel computes the inner product of two strings using the number of subsequences common to the two strings, rather than using the number of occurrences of such subsequences together with their degrees of contiguity.[1]

# 2 Project Focus

The primary goal of this project was to implement two kernels for use in support vector machines, SSK and SSK with lambda pruning (SSK-LP). As MATLAB's Bioinformatics Toolbox was used for its support vector machine, the kernels were originally implemented in MATLAB. Later, C++ implementations were developed due to concerns about the efficiency of the MATLAB kernels. These implementations include support for tuning parameters such as the weighting decay factor $\lambda$ and the substring length $n$, as well as the approximation factor $\theta$ for SSK-LP.

To allow for a comparison with the initial results obtained by Lodhi et al., the effects of these tuning parameters were explored using the Reuters dataset. Additionally, the actual speeds of the kernels were examined, and the relative accuracies of SSK and SSK-LP in support vector machines were compared.

The last part of the project involved exploring improvements to SSK-LP. A method of caching was developed for SSK-LP that did not significantly increase memory consumption over SSK, and the resulting approximated string subsequence kernel, SSK-LPC, was compared with SSK and SSK-LP without caching.

# 3 Implementation

MATLAB R2011b was used as the primary computing environment. The string kernels I implemented were then called as the kernel function of the support vector machine found in the Bioinformatics Toolbox. Good results were obtained using the default sequential minimal optimizer.

Lodhi et al. decreased the amount of computation performed by their support vector machine using a technique that limited the number of kernel evaluations needed. My focus was on optimizing the string kernel itself, so I used the more standard support vector machine found in MATLAB. Thus, our results may not be directly comparable.

To test the correctness of more efficient implementations of the string kernel, a straightforward implementation of SSK was first developed in MATLAB, followed by the recursive version given above. The resulting string subsequence kernel was too slow to run extensive experiments with. Due to the computational complexity of the algorithm, a more efficient C++ version was developed and linked as a MATLAB executable (MEX-file). The resulting kernel functions can be evaluated in about 0.2% of the time taken by the equivalent MATLAB functions. Sample timings illustrating the need for a compiled implementation are given in Table 5 in the Results section.

The original SSK function is implemented in ssk.cpp as a MEX-file. The results of the two intermediate functions used in the recursive computation, $K'$ and $K''$, are kept in a cache. For maximum efficiency, the parameters to SSK are set as global variables to limit overhead in function calling. Due to the recursive nature of the dynamic programming solution, the substrings that the two helper functions operate on always start at the beginnings of the strings proper, so it suffices to pass only the lengths of the substrings back and forth between the helper functions. Pre-computing all the powers of lambda that may be required seems to save a small amount of time. All of these steps could easily be done by a C++ class designed to provide a real-world string kernel.

SSK with lambda pruning is implemented in ssk_lp.cpp. The extra parameter $m$ represents the limit of recursion, corresponding to $\theta$. Unlike ssk.cpp, this kernel function does not store intermediate results. Using a cache that depended on the positions in the two strings, the length $n$ of the common substring to search for, and the recursion depth parameter $m$ would cause the memory requirements to quickly scale beyond what a typical computer offers, for modestly sized strings and fairly small values of $k$ and $m$. This means that the approximation becomes slower than the exact SSK algorithm when $\theta$, which determines the accuracy of the approximation, becomes too great. For example, SSK-LP loses its speed advantage over SSK when $n = 3$ and $\theta = 9n$; when $n = 4$ and $\theta = 6n$; when $n = 5$ and $\theta = 5n$. Furthermore, SSK-LP quickly becomes orders of magnitude slower than SSK when the parameters are

even higher, as Seewald and Kleedorfer report.[5] Runtime comparisons for these kernels appear in the Appendix.

## 3.1 SSK with Cached Lambda Pruning

It was desirable to include caching in SSK-LP without using significantly more memory than SSK, which motivated the design of SSK with lambda pruning and caching (SSK-LPC). Caching values of the helper functions is more difficult in SSK-LP because of the addition of the depth parameter *m*. The crucial observation is that the cached value of a function call need not correspond to the exact value of the depth parameter *m* to be useful in computation. Consider the value of $K'(s, t, i, m_1)$. If a cached value *c* exists for $K'(s, t, i, m_2)$, where $m_2 > m_1$, *v* represents the value of $K'(s, t, i)$ to a higher degree of recursion, so it is actually more accurate than the value of $K'(s, t, i, m_1)$ would be. Since returning a cache hit for any value of *m* that is equal to or greater than the value called for results in an approximation that is at least as close as what would have been computed after a cache miss, it is not necessary to retain separate cache versions for more than one value of *m*.

Thus, we can implement SSK with lambda pruning and caching. On a query to the cache for a set of parameters and a value of *m*, if an entry in the cache has been recorded for those same parameters and an equal or greater *m*-value, then that entry can be returned. Otherwise, a cache miss is reported, and once the approximate value is calculated, that value is stored in the cache location for those parameters.

This implementation requires two arrays for caching, for each helper function: one for the actual values of the helper functions, and one for the values of *m* for which each of those entries was calculated. The additional memory footprint need not be large, because the range of *m* will typically be small and will thus fit within a single byte for each entry, compared to the four or eight bytes used for storing the function values. Because of the check of the array of *m*-values, the larger array of function values need not be initialized, which further constrains the overhead from the extra caching. In contrast, exact caching for every value of *m* would increase memory consumption by $\theta$, which is itself a factor of *n*.

In my implementation of the SVM testing harness, the compiled string kernel is called once for each kernel evaluation needed by the SVM in MATLAB. This results in a large number of frequent, sizeable memory allocations and deallocations when training the SVM or otherwise calling the kernel function repeatedly, such as when running experiments to time the functions. To ensure that this continual memory allocation was not causing SSK or SSK-LPC to perform more slowly, similar C++ kernel functions were developed that accepted arrays of strings as input and returned an array of kernel values as output; these functions needed only allocate memory for the cache once for the entire set of kernel evaluations. Comparing the two implementations showed little difference in performance based on when memory was allocated.

# 4 Results

In the first set of experiments, SSK and SSK-LP were used in support vector machines to learn classifications of news articles. As the substring lengths $n$ and values of the decay parameter $\lambda$ were varied, the effects on the precision and recall of the classifiers were observed. These results are compared with those originally presented by Lodhi et al. The tests over different substring lengths used a constant value of $\lambda$ = 0.01, for which SSK-LP closely approximates SSK. In this setup, SSK-LP was shown to perform well.

The second set of experiments measured and compared the runtimes of SSK and the two approximations of SSK, identifying the parameters under which SSK-LP and SSK-LPC performed best.

## 4.1 Experimental Setup

The timings presented in Appendix 1 are relative to my system. These experiments were run on an Alienware M14x, with an Intel Core i7-2630QM 2.0 GHz processor, which ran at between 2.8 and 2.9 GHz. The laptop had eight gigabytes of DDR3-1333 RAM. The experiments were run within MATLAB R2011b (64-bit) in Windows 7, and the C++ files were compiled using MATLAB's MEX utility with the Microsoft Visual C++ 10 compiler.

These experiments used the same Reuters dataset used by Lodhi et al., Reuters-21578, in order to make similar comparisons. This is a set of stories from the Reuters news agency, currently available at http://daviddlewis.com/resources/testcollections/reuters21578. Due to the long computation times of SSK and SSK-LP, a subset of the "ModeApte" split was used, with the same number of training and testing examples of the categories used by Lodhi et al. The categories, along with the number of training and testing examples selected in each experiment, were as follows: corn (38, 10), crude (76, 15), acquisition (114, 25), earn (152, 40).

Parsing of the Reuters dataset is done in Python. This involves tidying up the input XML, parsing it for examples of the appropriate class, removing punctuation and excess whitespace from examples, and writing the examples to simple text files. These files are then read by MATLAB to randomly select the training and testing examples.

Performance is measured by the $F_1$ score, which is an equally weighted measure of the precision $p$ and recall $r$ of a binary classifier. The $F_1$ score is given as follows:

$$F = 2\frac{p * r}{p + r}$$

## 4.2 Effect of Varying Substring Length

Lodhi et al. conducted two sets of experiments to determine the effectiveness of a text categorization learning system when varying the parameters of SSK. They first show the effects of varying the substring length $n$, keeping the decay parameter $\lambda$ constant at 0.5. Since smaller values of $\lambda$ seem to be more effective on the Reuters dataset for both SSK and SSK-LP, I present the effects of

varying $n$ using a constant $\lambda$ of 0.01 in SSK-LP on the Reuters dataset. Table 1 and Table 2 present the results of SSK-LP on two splits of the data.

Using $\lambda$ = 0.01 with SSK-LP resulted in better performance than what Lodhi et al. reported using $\lambda$ = 0.5 with SSK. With such a small lambda, SSK-LP and SSK are virtually identical. However, some differences in our results may be attributed to the different support vector machines used in each study, so the results are not directly comparable.

For categories corn and crude, Lodhi et al. achieved the most accurate results with $n$ = 4 (.783) and $n$ = 5 (.936), respectively. My results are similar, with SSK-LP the most accurate at $n$ = 4 for both categories, although the peak $F_1$ scores were higher (.981 and .990). Lodhi et al. reported peak precisions at $n$ = 3 for corn and $n$ = 6 for crude; in contrast, my precision for corn was 1.0 except at $n$ = 3, and peak precision for crude occurred at $n$ = 4. Both results showed relatively low recall for corn except around $n$ = 4, although my recall for crude was consistently around 1.0 while recall for crude in Lodhi et al. peaked at $n$ = 5. Overall, in both sets of experiments, SSK/SSK-LP performed well with fairly short substring lengths.

For categories acq and earn, results did not drop off as dramatically at higher substring length. In fact, SSK-LP was slightly more accurate at $n$ = 8, where precision peaked for earn, and recall peaked for acq. The results of Lodhi et al. showed better results around $n$ = 6, where precision and recall peaked for both earn and acq. This shows that higher values of $n$ can be useful, although the difference in accuracies was not as dramatic as in the corn/crude split, where more accurate results were obtained at lower substring lengths.

| Category | Kernel | Length | F1 | | Precision | | Recall | |
|---|---|---|---|---|---|---|---|---|
| | | $n$ | Mean | SD | Mean | SD | Mean | SD |
| corn | SSK-LP | 3 | 0.960 | 0.039 | 0.985 | 0.034 | 0.942 | 0.076 |
| | | **4** | **0.981** | 0.041 | 1.000 | 0.000 | 0.967 | 0.075 |
| | | 5 | 0.972 | 0.048 | 1.000 | 0.000 | 0.950 | 0.087 |
| | | 6 | 0.902 | 0.047 | 1.000 | 0.000 | 0.825 | 0.083 |
| | | 7 | 0.875 | 0.060 | 1.000 | 0.000 | 0.783 | 0.090 |
| | | 8 | 0.753 | 0.065 | 1.000 | 0.000 | 0.608 | 0.086 |
| | | 10 | 0.685 | 0.071 | 1.000 | 0.000 | 0.525 | 0.083 |
| crude | SSK-LP | 3 | 0.976 | 0.023 | 0.965 | 0.045 | 0.989 | 0.025 |
| | | **4** | **0.990** | 0.023 | 0.980 | 0.044 | 1.000 | 0.000 |
| | | 5 | 0.984 | 0.027 | 0.971 | 0.051 | 1.000 | 0.000 |
| | | 6 | 0.946 | 0.026 | 0.898 | 0.048 | 1.000 | 0.000 |
| | | 7 | 0.933 | 0.025 | 0.876 | 0.044 | 1.000 | 0.000 |
| | | 8 | 0.885 | 0.023 | 0.795 | 0.037 | 1.000 | 0.000 |
| | | 10 | 0.864 | 0.021 | 0.761 | 0.033 | 1.000 | 0.000 |

**Table 1: The performance of SVM with SSK-LP for Reuters categories corn and crude, varying the subsequence length. The results are averaged across runs over 12 different random subsets of the data.**

| Category | Kernel | Length | F1 | | Precision | | Recall | |
|---|---|---|---|---|---|---|---|---|
| | | $n$ | Mean | SD | Mean | SD | Mean | SD |
| acq | SSK-LP | 3 | 0.957 | 0.027 | 0.932 | 0.043 | 0.983 | 0.020 |
| | | 4 | 0.966 | 0.024 | 0.947 | 0.040 | 0.987 | 0.019 |
| | | 5 | 0.969 | 0.018 | 0.950 | 0.034 | 0.990 | 0.017 |
| | | 6 | 0.968 | 0.021 | 0.948 | 0.039 | 0.990 | 0.017 |
| | | 7 | 0.969 | 0.018 | 0.950 | 0.034 | 0.990 | 0.017 |
| | | **8** | **0.971** | 0.019 | 0.948 | 0.039 | 0.997 | 0.011 |
| | | 10 | 0.959 | 0.029 | 0.944 | 0.042 | 0.977 | 0.034 |
| earn | SSK-LP | 3 | 0.971 | 0.018 | 0.989 | 0.013 | 0.954 | 0.030 |
| | | 4 | 0.978 | 0.017 | 0.992 | 0.012 | 0.965 | 0.028 |
| | | 5 | 0.980 | 0.012 | 0.994 | 0.011 | 0.967 | 0.024 |
| | | 6 | 0.979 | 0.014 | 0.994 | 0.011 | 0.965 | 0.028 |
| | | 7 | 0.980 | 0.012 | 0.994 | 0.011 | 0.967 | 0.024 |
| | | **8** | **0.981** | 0.014 | 0.998 | 0.007 | 0.965 | 0.028 |
| | | 10 | 0.974 | 0.019 | 0.986 | 0.021 | 0.963 | 0.030 |

**Table 2:** **The performance of SVM with SSK-LP for Reuters categories acq and earn, varying the subsequence length. The results are averaged across runs over 12 different random subsets of the data.**

## 4.3 Effect of Varying Weight Decay Parameter

The weight decay parameter $\lambda$ controls how non-contiguous subsequences are weighted, with higher values resulting in higher weights for less contiguous subsequences, and lower values penalizing gaps to a larger degree. These experiments, mirroring those of Lodhi et al., test the effectiveness of text categorization using SSK with different values of the lambda parameter. Because SSK-LP is less accurate at higher values of $\lambda$, the exact SSK algorithm was used here. In accordance with Lodhi et al., a constant value of $n$ = 5 was chosen. Results are presented in Table 3 and Table 4.

Again, overall, the accuracies with which the MATLAB support vector machine using SSK classified the test samples were higher than those reported by Lodhi et al. Since the test conditions were similar in this case, the difference is probably attributable to the different support vector machines used.

I encountered little variation in accuracies as $\lambda$ varied from 0.01 to 0.5. Higher values of $\lambda$ would have been expected to yield poor results, as shown by Lodhi et al. Their accuracies peaked at $\lambda$ = 0.01 and $\lambda$ = 0.3 for categories corn and crude, and dropped off above $\lambda$ = 0.5; for categories acq and earn, they saw less variation.

| Category | Kernel | Decay $\lambda$ | F1 | | Precision | | Recall | |
|---|---|---|---|---|---|---|---|---|
| | | | Mean | SD | Mean | SD | Mean | SD |
| corn | SSK | **0.01** | **0.955** | 0.065 | 0.981 | 0.041 | 0.933 | 0.094 |
| | | 0.1 | 0.946 | 0.062 | 0.981 | 0.041 | 0.917 | 0.090 |
| | | 0.3 | 0.946 | 0.062 | 0.981 | 0.041 | 0.917 | 0.090 |
| | | 0.5 | 0.946 | 0.062 | 0.981 | 0.041 | 0.917 | 0.090 |
| crude | SSK | **0.01** | **0.973** | 0.039 | 0.960 | 0.057 | 0.989 | 0.025 |
| | | 0.1 | 0.968 | 0.037 | 0.949 | 0.055 | 0.989 | 0.025 |
| | | 0.3 | 0.968 | 0.037 | 0.949 | 0.055 | 0.989 | 0.025 |
| | | 0.5 | 0.968 | 0.037 | 0.949 | 0.055 | 0.989 | 0.025 |

**Table 3:** **The performance of SVM with SSK for Reuters categories corn and crude, varying the weight decay parameter. The results are averaged across runs over 6 different random subsets of the data.**

| Category | Kernel | Decay $\lambda$ | F1 | | Precision | | Recall | |
|---|---|---|---|---|---|---|---|---|
| | | | Mean | SD | Mean | SD | Mean | SD |
| acq | SSK | 0.01 | 0.966 | 0.019 | 0.934 | 0.035 | 1.000 | 0.000 |
| | | 0.1 | 0.977 | 0.019 | 0.956 | 0.036 | 1.000 | 0.000 |
| | | 0.3 | 0.977 | 0.019 | 0.956 | 0.036 | 1.000 | 0.000 |
| | | 0.5 | 0.973 | 0.015 | 0.948 | 0.030 | 1.000 | 0.000 |
| earn | SSK | 0.01 | 0.977 | 0.013 | 1.000 | 0.000 | 0.955 | 0.024 |
| | | 0.1 | 0.985 | 0.013 | 1.000 | 0.000 | 0.970 | 0.024 |
| | | 0.3 | 0.985 | 0.013 | 1.000 | 0.000 | 0.970 | 0.024 |
| | | 0.5 | 0.982 | 0.010 | 1.000 | 0.000 | 0.965 | 0.020 |

**Table 4:** **The performance of SVM with SSK for Reuters categories acq and earn, varying the weight decay parameter. The results are averaged across runs over 5 different random subsets of the data.**

## 4.4 Runtime Measurements

First, timings are presented to compare the MATLAB and C++ implementations of SSK using fairly small string lengths. The MATLAB implementations are remarkably slower, and were not used in any other experiments.

| $\|s\| = \|t\|$ | $n$ | MATLAB (seconds) | C++ (seconds) | Improvement factor |
|---|---|---|---|---|
| 100 | 3 | 0.41 | 0.0009 | 456 |
| 100 | 4 | 0.60 | 0.0012 | 506 |
| 100 | 5 | 0.78 | 0.0015 | 521 |
| 200 | 3 | 1.94 | 0.0037 | 521 |
| 200 | 4 | 2.91 | 0.0055 | 530 |
| 200 | 5 | 3.89 | 0.0079 | 491 |
| 300 | 3 | 4.64 | 0.0103 | 452 |
| 300 | 4 | 6.95 | 0.0153 | 453 |
| 300 | 5 | 9.26 | 0.0208 | 444 |
| 400 | 3 | 8.38 | 0.0194 | 432 |
| 400 | 4 | 12.63 | 0.0301 | 419 |
| 400 | 5 | 16.79 | 0.0408 | 411 |
| 500 | 3 | 13.38 | 0.0330 | 405 |
| 500 | 4 | 20.08 | 0.0517 | 389 |
| 500 | 5 | 26.80 | 0.0691 | 388 |

**Table 5: The performance of the MATLAB and C++ implementations of SSK with various parameters. Data is averaged over 5 runs.**

The compiled kernel functions were evaluated on strings from the Reuters dataset to investigate their running times. Twenty articles from the corn/crude split with at least 1,500 characters were chosen at random, and this set was randomly permuted to provide two orderings of the 20 strings. For each set of parameters (kernel function, string length, $n$ and $\theta$), five trials were run. In each trial, the kernel was run 20 times, once for each string from the first ordering (which was paired with the corresponding string in the second ordering). The results were then averaged across all 100 executions to provide the average time of the given kernel with the given parameter values over the string pairs.

The timings for the kernels are plotted in Appendix. The substring length $n$ ranged from 2 to 7; $\theta$ ranged from $3n$ to $6n$. The performance of SSK-LP relative to SSK is similar to what was found by Seewald and Kleedorfer. Their results showed SSK-LP performing several orders of magnitude more slowly than SSK for values of $\theta$ from $9n$ to $12n$, so I instead included timings for SSK-LPC at those values of $\theta$ to show that, while SSK-LPC still performed worse than SSK at such recursive depths, it scales better with higher $\theta$ than SSK-LP.

Generally, for low values of $\theta$, SSK-LPC does not outperform SSK-LP, due to the overhead inherent in caching. For higher values of $\theta$, SSK-LPC is faster than SSK-LP, but both SSK-LP and SSK-LPC take longer than SSK. However, good results for SSK-LPC are obtained for some parameter combinations. Table 6 gives the performance of SSK-LPC relative to that of SSK-LP for various values of $n$ and $\theta$. The results show that SSK-LPC performs better for higher values of these parameters. SSK-LPC is then compared with SSK in Table 7. While SSK-LPC is still slower than SSK for higher values of $\theta$, there is a region in which SSK-LPC is at least somewhat faster than both SSK and SSK-LP.

| | $n$ = 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| $\theta$ = 3$n$ | 2.30 | 2.57 | 2.55 | 2.38 | 2.13 | 1.83 |
| 4$n$ | 2.00 | 1.86 | 1.57 | 1.16 | **0.846** | **0.594** |
| 5$n$ | 1.76 | 1.48 | **0.944** | **0.570** | **0.335** | **0.187** |
| 6$n$ | 1.63 | 1.12 | **0.584** | **0.284** | **0.130** | **0.055** |

**Table 6:** Ratios of time for SSK-LPC to time for SSK-LP, with |s| = |t| = 1500, averaged across 20 pairs of strings.

| | $n$ = 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| $\theta$ = 3$n$ | 0.185 | 0.179 | 0.187 | 0.203 | 0.212 | 0.223 |
| 4$n$ | 0.252 | 0.297 | 0.370 | 0.427 | **0.486** | **0.556** |
| 5$n$ | 0.325 | 0.482 | **0.613** | **0.773** | **0.929** | 1.13 |
| 6$n$ | 0.418 | 0.677 | **0.922** | 1.22 | 1.52 | 1.90 |

**Table 7:** Ratios of time for SSK-LPC to time for SSK, with |s| = |t| = 1500, averaged across 20 pairs of strings. Bolded items are instances where SSK-LPC outperformed both SSK and SSK-LP.

As the number of characters in the strings increases, both SSK-LP and SSK-LPC run more quickly relative to SSK. Meanwhile, the relative performance of SSK-LPC to SSK-LP declines somewhat, probably due to the increase in memory consumption of SSK-LPC. Two representative charts are presented below.

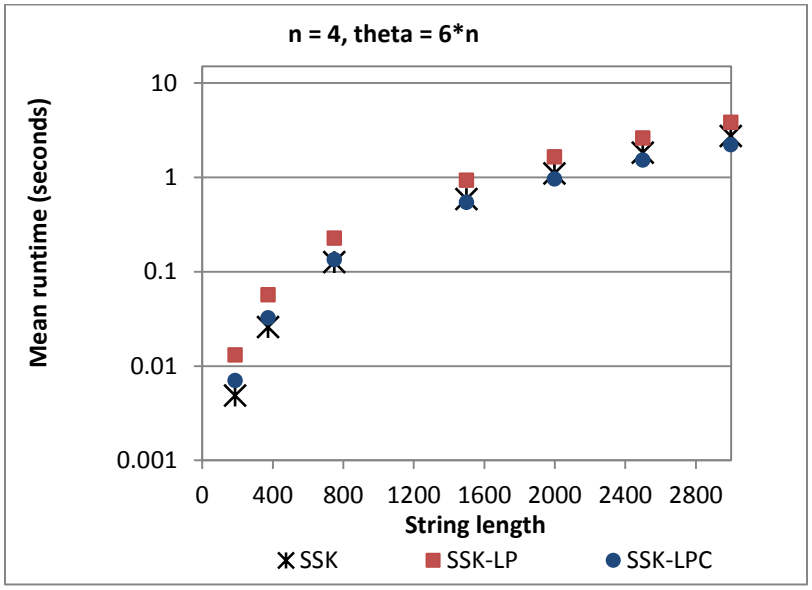



**Figure 1:** Runtimes of SSK, SSK-LP and SSK-LPC. As string length increases, relative performances of SSK-LP and SSK-LPC improve. Results were averaged over 10 pairs of strings.

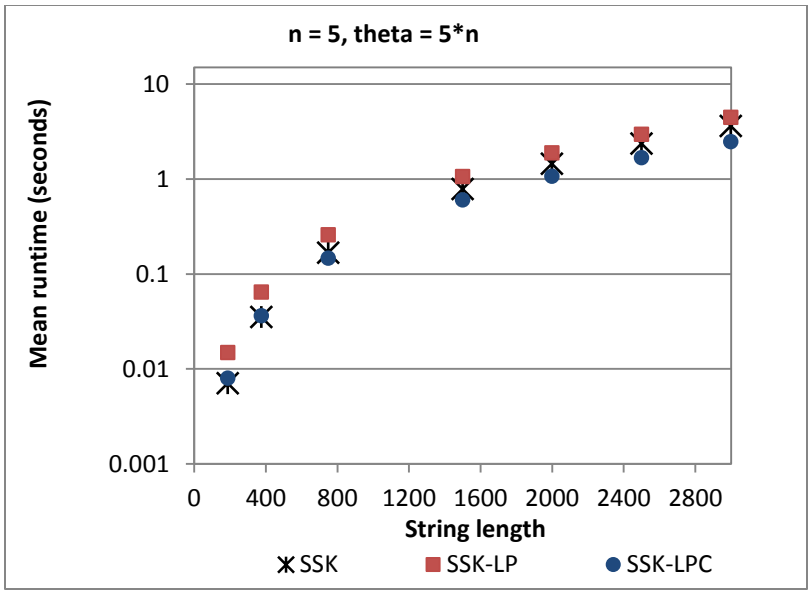



Figure 2: Runtimes of SSK, SSK-LP and SSK-LPC. Results were averaged over 10 pairs of strings.

# 5 Conclusions and Future Work

SSK remains computationally expensive, and SSK with lambda pruning only offers a fast approximation for a subset of the parameter space. The option to use caching in SSK with lambda pruning allows for its use with more parameters by providing cheaper approximations in certain cases, which also allows a higher value of the weight decay parameter to be used.

With lambda pruning and potentially caching and lambda pruning, SSK is still a slow algorithm. More work could be done to find more efficient implementations of SSK. SSK should also be more extensively compared with other string kernels that have been successfully used in other learning problems for a better understanding of the contexts in which SSK offers advantages over other string kernels.

# References

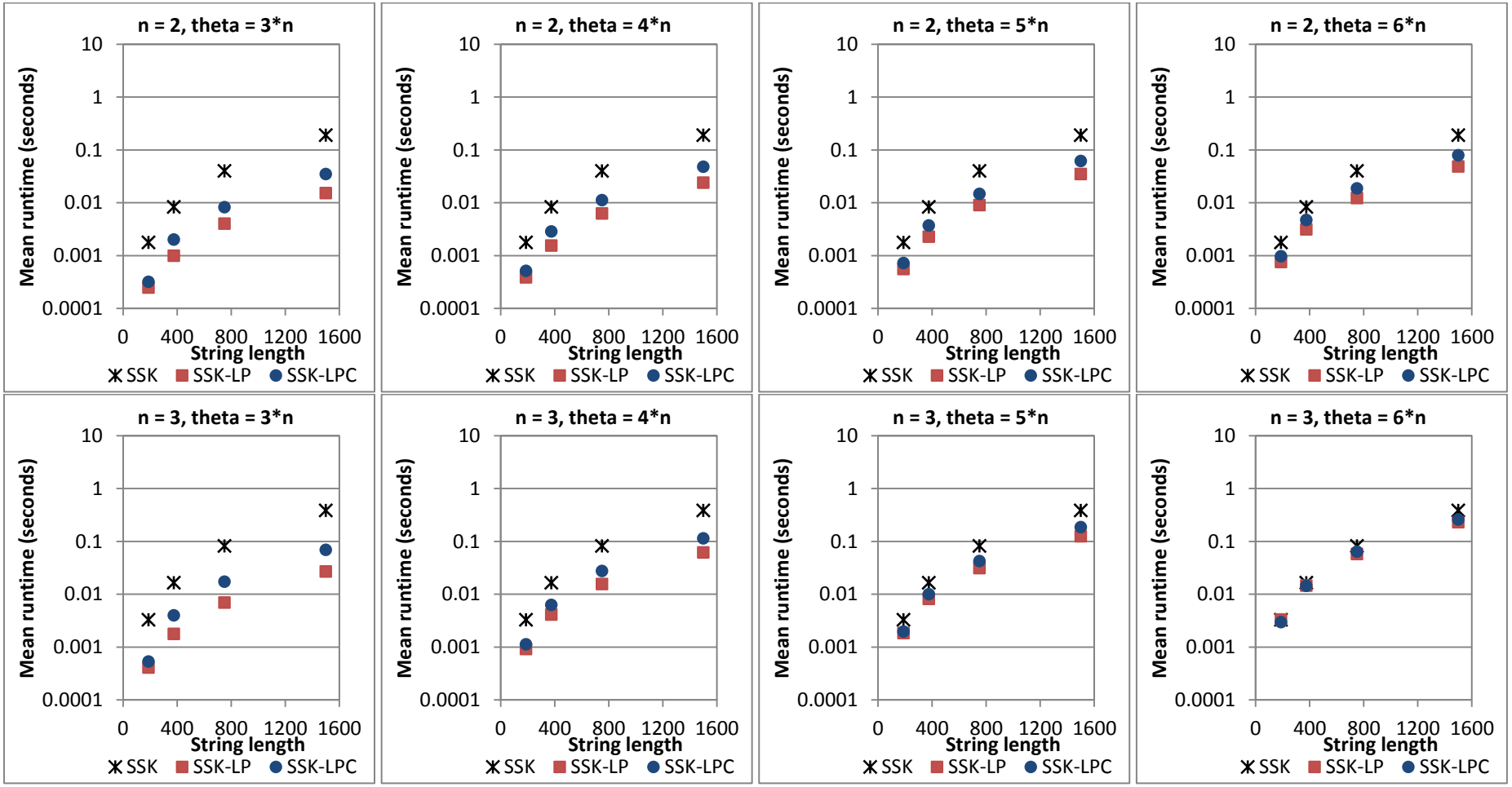
1. Kontorovich, L., et al. Learning linearly separable languages. *ALT 2006*, 288-303, 2006.
2. Leslie, C., et al. The spectrum kernel: A string kernel for SVM protein classification. *Pacific Symposium on Biocomputing*, 7:566-575, 2002.
3. Lodhi, H., et al. Text classification using string kernels. *Journal of Machine Learning Research*, 2:419-444, 2002.
4. Sculley, D., et al. Spam filtering using inexact string matching in explicit feature space with on-line linear classifiers. *TREC 2006*, 2006.
5. Seewald, A., and Kleedorfer, Florian. Lambda pruning: an approximation of the string subsequence kernel. *Advances in Data Analysis and Classification*, 1(3):221-239, 2007.

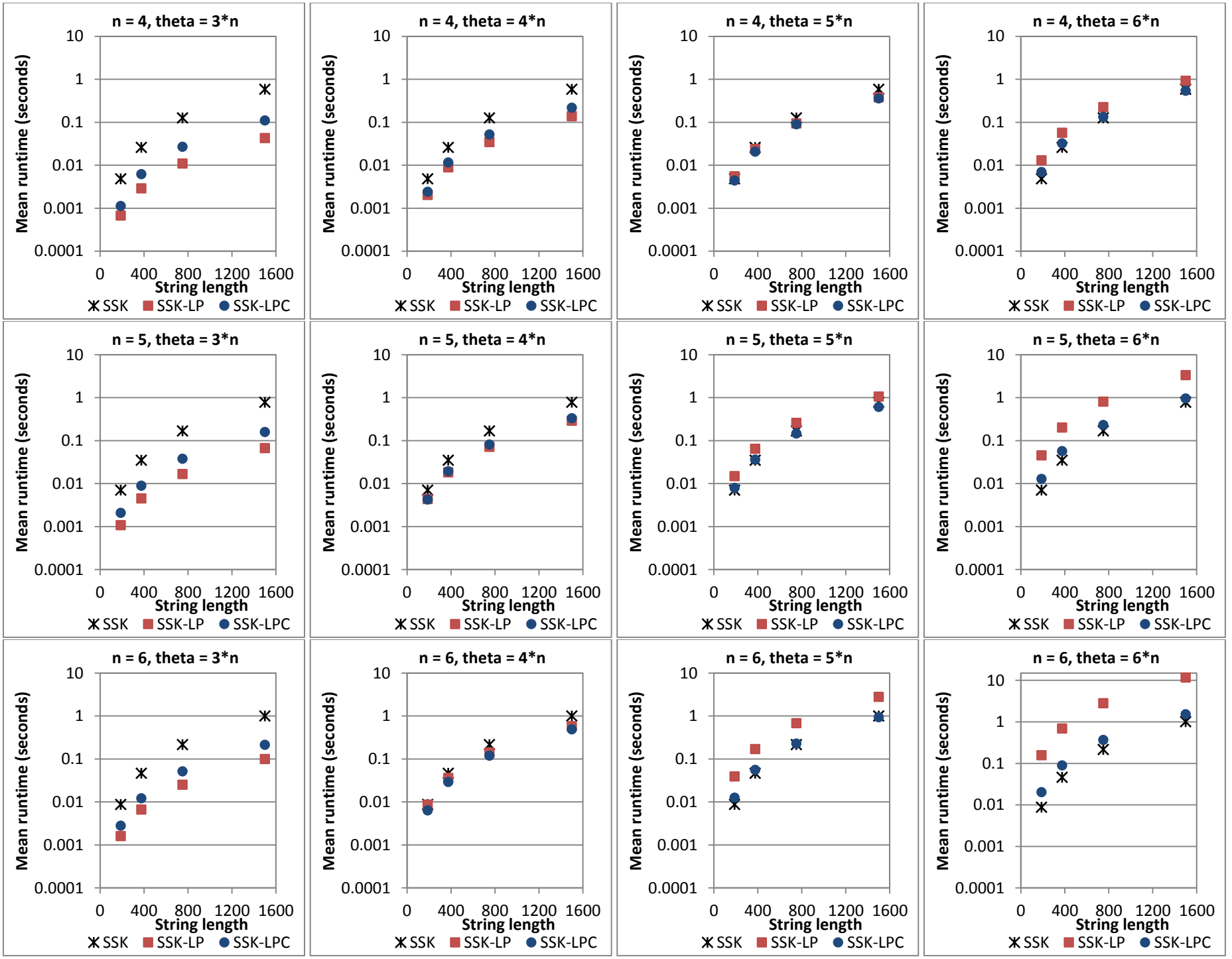
# Appendix

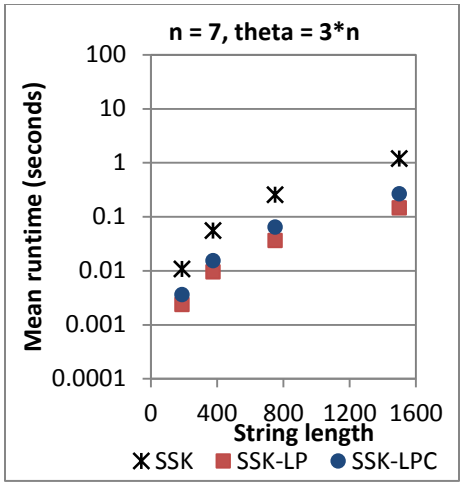
## Runtime Comparisons

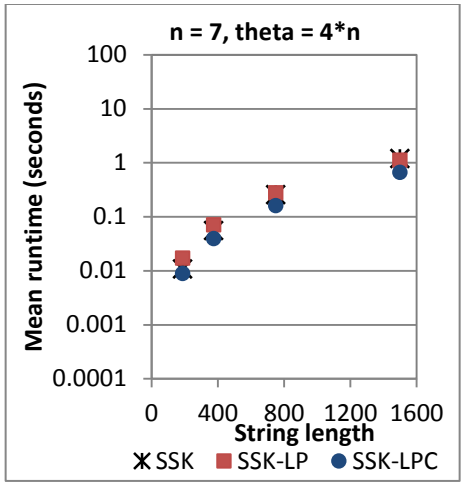
The string subsequence kernels were evaluated on strings from the Reuters dataset to investigate their running times. Twenty articles from the corn/crude split with at least 1,500 characters were chosen at random, and this set was randomly permuted to provide two orderings of the 20 strings. For each set of parameters (kernel function, string length, $n$ and $\theta$), five trials were run. In each trial, the kernel was run 20 times, once for each string from the first ordering (which was paired with the corresponding string in the second ordering). The results were then averaged across all 100 executions to provide the average time of the given kernel with the given parameters.

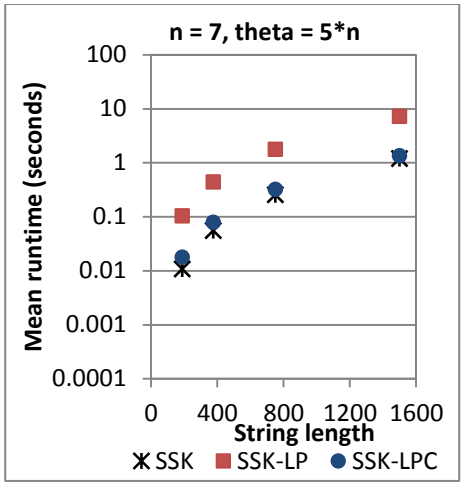
The following are the results for $n$ from 2 to 7, and $\theta$ from $3n$ to $6n$. Results for higher values of $\theta$ follow.

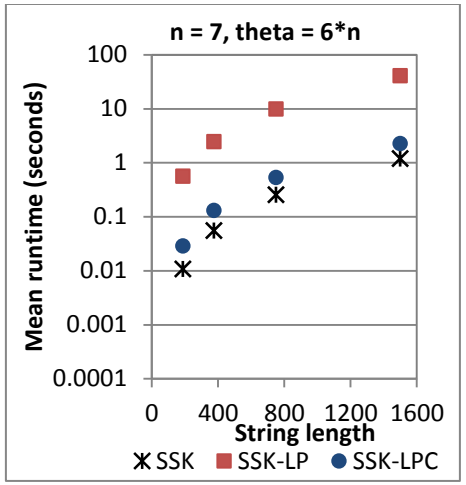



n = 4, theta = 3*n
n = 4, theta = 4*n
n = 4, theta = 5*n
n = 4, theta = 6*n
n = 5, theta = 3*n
n = 5, theta = 4*n
n = 5, theta = 5*n
n = 5, theta = 6*n
n = 6, theta = 3*n
n = 6, theta = 4*n
n = 6, theta = 5*n
n = 6, theta = 6*n
Mean runtime (seconds)
10
1
0.1
0.01
0.001
0.0001
0
400
800
1200
1600
String length
SSK
SSK-LP
SSK-LPC



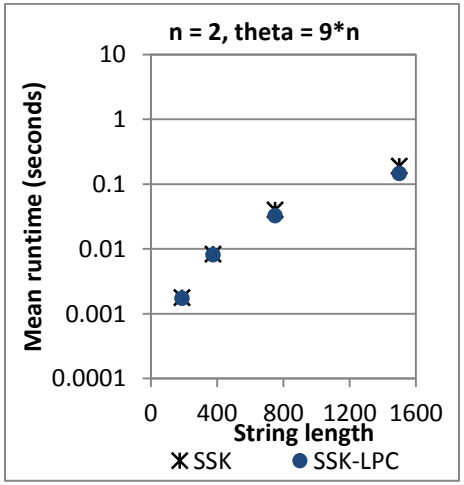



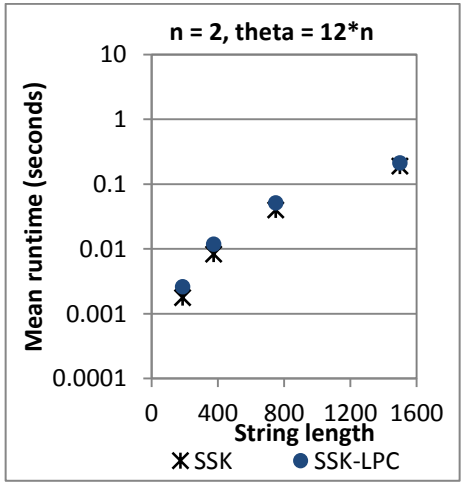



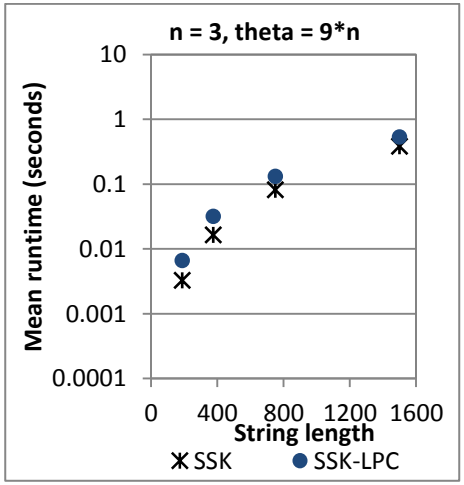



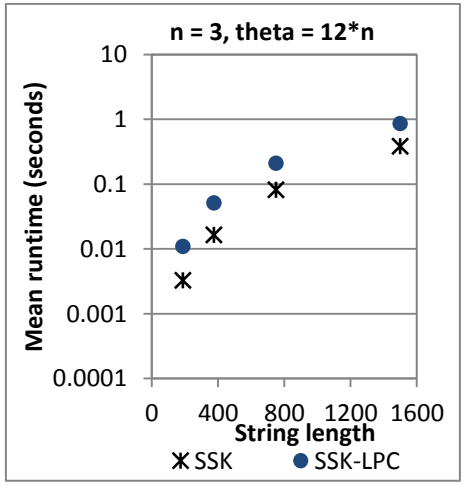

For higher values of $\theta$, SSK-LP was significantly slower than the other kernels, so SSK-LP was excluded from the remaining trials. These results are included to show that SSK-LPC does not become intractably slow as quickly as SSK-LP.

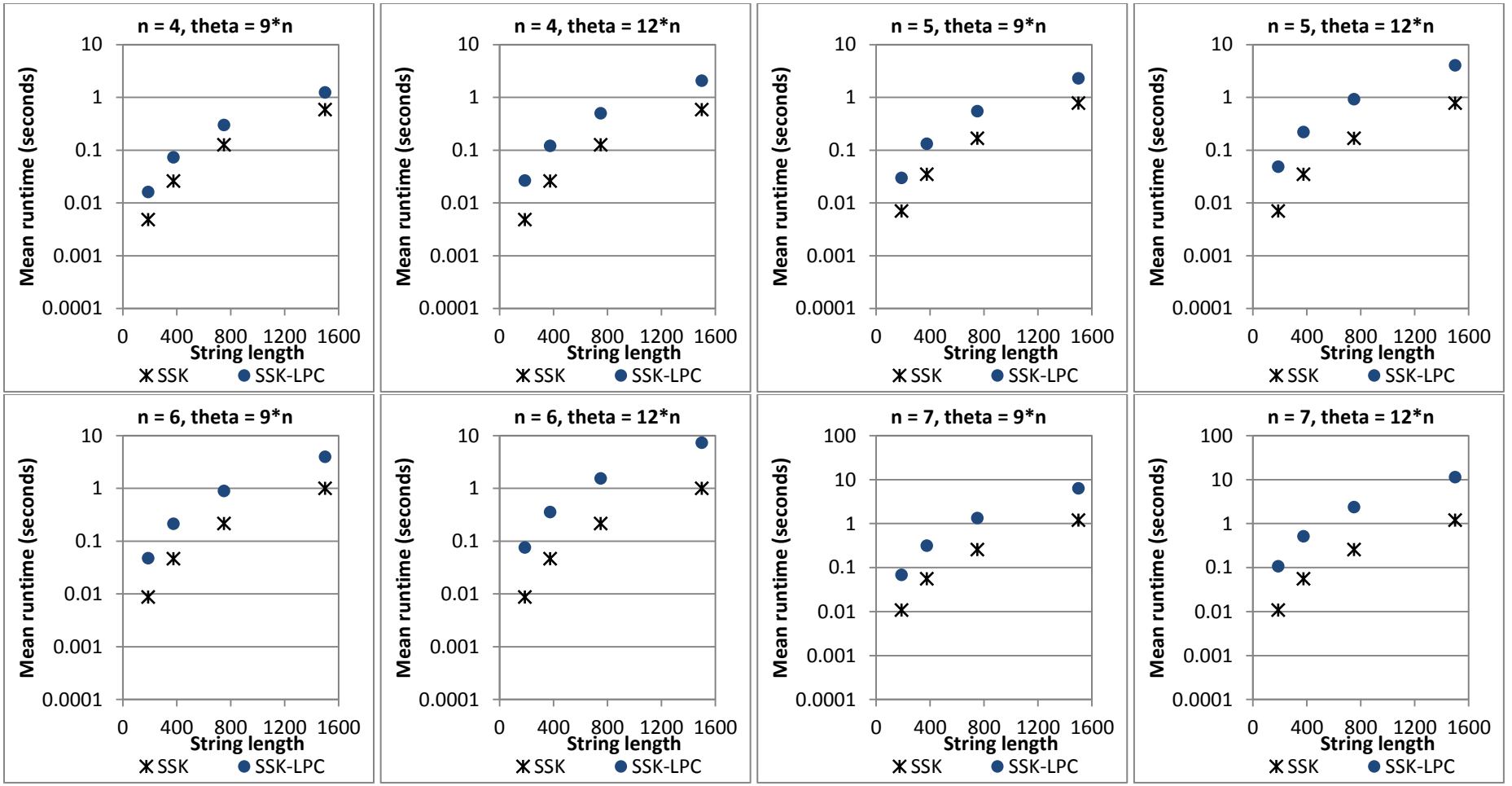
Timings for SSK-LP compared to SSK for these values of $\theta$ can be found in Seewald and Kleedorfer's results.




n = 4, theta = 9*n
n = 4, theta = 12*n
n = 5, theta = 9*n
n = 5, theta = 12*n
n = 6, theta = 9*n
n = 6, theta = 12*n
n = 7, theta = 9*n
n = 7, theta = 12*n
Mean runtime (seconds)
String length
0
400
800
1200
1600
SSK
SSK-LPC