Machine Learning

Roberto Fernandez
Collaborated with Kevin Tse
Problem Set 5

Exercise 1

1. This kernel is measuring the similarity between two strings by summing the product of the number of occurrences of each substring in $\overline{a}, \overline{b}$ over all such possible substrings of length κ. We can thus see, intuitively, that this sum will have much higher values for strings that are similar, in the sense that they share many of the same characters or sequences of characters.

   We know that it is positive semi-definite because this kernel is actually the inner product of the function mapping: $\varphi \colon \Sigma^* = \bigcup_{k=1}^{|\Sigma|} \Sigma^k \to \mathbb{R}^{\Sigma^n}$ such that $[\varphi(\overline{a})]_u = n_u(\overline{a})$ such that the fact that this is a Mercer Kernel is obvious. We can then find the value of this kernel for contiguous substrings as follows:

Algorithm 1 Contiguous Substring Kernel

```
sum ← 0
for i = 0 to i = ℓ_a − κ do
    s̄ ← ā_{i:i+κ}
    for j = 0 to j = ℓ_b − κ do
        t̄ ← b̄_{j:j+κ}
        if s̄ == t̄ then
            sum ← sum + 1
    return sum
```

2. This kernel has a meaning similar to the kernel above but in this case we are considering subsequences of the given strings instead of substrings, so this is in a way a looser form of the above kernel where we don't care about any intermediate characters as long as we can find our subsequence in the string (once we add the λ parameter we penalize incorrect characters). This is thus clearly a valid semi positive definite kernel due to the same reason as above except our feature mapping is now a different numerical value.

   (a) For this part of the problem we only consider single members of our alphabet and we only care about characters that can be found in both strings so we will basically count the number of occurrences of each character in $\overline{a}$ and then traverse $\overline{b}$ calculating the product $n_s(\overline{a})n_s(\overline{b})$ using the previously computed values and find the running sum. Note that the assignment of c is basically a hash which, given $\Sigma = \{a_1, a_2, \ldots, a_k\}$ will map $a_i \to i$: (on next page, algo #2)

   (b) It is clear that the value of this kernel, since we are looking for substrings of size p inside of two strings of size p, will only be non-zero (and 1 in the non-zero case...) if both strings are equal. We thus have the following algorithm: (algo #3)

Algorithm 2 Single Character

for $i = 0$ to $i = \ell_a - 1$ **do**
    $c \leftarrow a[i] -' a'$
    $vals_a[c] \leftarrow vals_a[c] + 1$
for $i = 0$ to $i = \ell_b - 1$ **do**
    $c \leftarrow b[i] -' a'$
    $vals_b[c] \leftarrow vals_b[c] + 1$
$sum \leftarrow 0$
for $i = 0$ to $i = |\Sigma|$ **do** $sum \leftarrow sum + vals_a[i] * vals_b[i]$
return sum

Algorithm 3 String Equality

for $i = 0$ to $i = p$ **do**
    if $a[i] != b[i]$ **then return** 0
return 1

(c) i. These precomputed values follow a dynamic programming approach and we can thus see that the only values missing when we consider the string $\overline{a}_{0:N+1}$ (i.e. consider the next character in the string) the only possible differences it can make to our kernel is if this adds oe of the missing characters for a previously 'ignored' subsequence. This is equivalent to saying that for some precomputed kernel value of $\kappa - 1$ length substrings we were only missing the last correct character in order to add it to k_κ and we thus traverse all these possibilities and add if necessary to get:

$$k_\ell(\overline{a}_{0:N+1}, \overline{b}_{0:j}) = k_\ell(\overline{a}_{0:N}, \overline{b}_{0:j}) + \sum_{k=\ell}^{j} [[a_{N+1} = b_k]] k_{\ell-1}(\overline{a}_{0:N}, \overline{b}_{0:k-1})$$

ii. This is exactly the same process as above but we are instead extending the information for $\overline{b}$ subsequence matches so we flip the indices to get:

$$k_\ell(\overline{a}_{0:i}, \overline{b}_{0:M+1}) = k_\ell(\overline{a}_{0:i}, \overline{b}_{0:M+1}) + \sum_{k=\ell}^{i} [[a_i = b_{M+1}]] k_{\ell-1}(\overline{a}_{0:i}, \overline{b}_{0:M+1})$$

iii. We notice that by memoizing not only the values of i, j but also κ we can avoid calculating the redundant parts of the kernel (going over parts of the string we know will contribute nothing to the kernel's value) and we thus get the following algorithm: which gives us an algorithm of $O(\kappa \ell_a \ell_b^2)$ complexity but we note that

Algorithm 4 Gappy Substring Kernel

for $k = 0$ to $k = \kappa$ **do**
    for $j = 0$ to $j = \ell_a$ **do**
        for $i = 0$ to $i = \ell_b$ **do**
            $t[i+1, j, k] \leftarrow t[i, j, k] + \sum_{s=p}^{j} [[a_{i+1} = b_s]] * t[i, s-1, k-1]$
            $t[i, j+1, k] \leftarrow t[i, j, k] + \sum_{s=p}^{j} [[a_{i+1} = b_s]] * t[s-1, j, k-1]$

there is overlap in the calculation of the kernel, namely that if we let S denote the inner sum we have that $S(\overline{a}_{0:i+k}, \overline{b}_{0:j+N}) = S(\overline{a}_{0:i+k}, \overline{b}_{0:j})$ if $\overline{a}_{i:i+k}$ is not found in $\overline{b}_{j:j+N}$ and that

$$S_p(\overline{a}_{0:i+k}, \overline{b}_{0:j+k}) = S_p(\overline{a}_{0:i+k}, \overline{b}_{0:j}) + k_{p-1}(\overline{a}_{0:i}, \overline{b}_{0:j})$$

so we can simplify our algorithm to $O(\kappa \ell_a \ell_b)$ time complexity.

3. This kernel calculation is similar to that done above but we have different functions that we call recursively since we are basically weighting our number functions by some number that decreases said weight as the length of the subsequence increases (i.e. it is more sparse in the input string).

   (a) For this part of the problem we take a similar approach as above but consider λ as a weight factor when adding but since $\kappa = 1$ we have that λ is simply a weighting factor for each entry so we multiply each part of our sum in the previously given algorithm by λ^2.

   (b) In this case it is clear that there cannot be any gaps since the size of our string is the same as the size of any substring so the factor of λ^2 from part (i) still remains and we use the same process as above and multiply by this factor.

   (c) Here we note that

   $$k_n(\overline{a}, \overline{b}) = \sum_{s \in \Sigma^\kappa} \sum_{i:\ s=\overline{a}[i]} \sum_{j:\ s=\overline{b}[j]} \lambda^{|\overline{a}|+|\overline{b}|-i_0-j_0+2}$$

   and we can thus apply the algorithms as above and instead multiply by the corresponding λ scalar.

4. Here we have that k_Σ measures the similarity between different elements of the alphabet itself and if we thus take $\prod_{i=1}^{\kappa} k_\Sigma(\overline{a}_i, \overline{b}_i)$ we are taking a similar similarity measure to that above since it is the distance between a substring of each string so it is basically a different form of a λ factor where we multiply by a weight defined by the kernel on the alphabet.

   With this being the case it is clear that it is positive semi-definite since it is just the inner product of a feature map, as shown above, and similar algorithms can be run simply multiplying by our k_Σ which is done in constant time and thus doesn't affect the complexity.

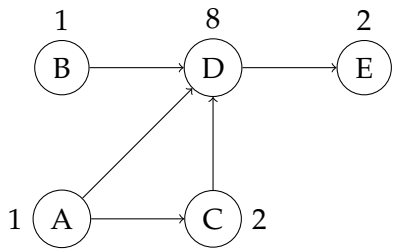
Exercise 2

1. We have $P(X_A, X_B, X_C) = P(X_C)P(X_A \mid X_C)P(X_B \mid X_C)$ and it is thus clear that $X_A \perp\!\!\!\perp X_B \mid X_C$ since our distribution gives us that $P(X_B \mid X_A, X_C) = P(x_B \mid X_C)$ and we can see that $X_A \not\perp\!\!\!\perp X_B$ because there is a flow of information from X_A to X_B or vice-verse. A counterexample that shows this is $X_C \in \{0, 1\}$ with $X_A, X_B = X_C + t$ where t is a random variable also in $\{0, 1\}$ and is chosen randomly for both A, B.

2. We have $P(X_A, X_B, X_C) = P(X_A)P(X_B)P(X_C \mid X_A, X_B)$ and it is thus clear that $X_A \perp\!\!\!\perp X_B$ since our distribution gives us that $P(X_A, X_B) = P(X_A)P(X_B)$ and we can also see that $X_A \not\perp\!\!\!\perp X_B \mid X_C$ because there is a flow of information from X_A to X_B if X_C is observed. A counterexample that shows this is similar to above with $X_A, X_B \in \{0, 1\}$ both randomly independently chosen and that $X_C = X_A + X_B$.

Exercise 3

We have the following Bayes net:



which can be fully expressed using $2_A^0 + 2_B^0 + 2_C^1 + 2_D^3 + 2_E^1 = 1_A + 1_B + 2_C + 8_D + 2_E = 14$ parameters.

Exercise 4

1. It is clear that $E \perp\!\!\!\perp G \mid F$ is true because the only path connecting E, G is $E \to F \to G$ and since F is given then no information can flow through the path (E only gave information through F but if F is given that information is now worthless.)
2. It is clear that $A \not\perp\!\!\!\perp C \mid B, G$ because we have the path $A \to D \to F \leftarrow E \leftarrow C$ which provides a flow of information such that G gives us information about A, C.
3. It is clear that $B \perp\!\!\!\perp C$ by definition since they are both root nodes with no conditionals observed.
4. It is clear that $A \not\perp\!\!\!\perp B \mid C, G$ because the path $A \to D \leftarrow B$ gives us information about A, B since G is a descendant of D.

Exercise 5

We have the joint distribution: $P(A, B, C, D, E) = P(A)P(E)P(B \mid A)P(D \mid A, E)P(C \mid B, D)$ and we get the following independency statements: $A \perp\!\!\!\perp E; B \perp\!\!\!\perp E; B \perp\!\!\!\perp D \mid A; A \perp\!\!\!\perp C \mid B, D; C \perp\!\!\!\perp E \mid D$. It is clear that every other combination is dependent since there exists a path through which information can flow.