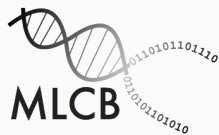


Kernels for proteins and MMD.

Philip Hartout

March 3, 2022



DBSSE

ETH zürich

- Here we want to go through the maths behind the kernels to make sure our implementation is efficient.
- Linear, Weissfeiler-Lehmann kernel

- Not much to optimize there.

Weisfeiler-Lehmann kernel

- Computing $\phi(G)$ needs to be done explicitly and can be done independently (and in parallel) prior to computing $K_{WL} = \phi(G)^T \phi(G')$
- How to compute $\phi(G)$? `networkx` has a function called `weisfeiler_lehman_subgraph_hashes`.
- Since we don't care about the order in the resulting $K_{WL} = \phi(G)^T \phi(G')$, we can list each product that needs to be done and execute them in parallel as well.

Weisfeiler-Lehmann kernel

What does the implementation look like?

```
def set_weisfeiler_lehman_hashes(  
    self, graph_type: str, n_iter: int  
) -> None:  
    hashes = dict(  
        Counter(  
            flatten_lists(  
                list(  
                    nx.weisfeiler_lehman_subgraph_hashes(  
                        self.graphs[graph_type],  
                        node_attr="residue",  
                        iterations=n_iter,  
                    ).values()  
                )  
            )  
        )  
    )  
    self.descriptors[graph_type]["weisfeiler-lehman-hist"] = hashes
```

Figure 1: Setting the hash histogram for each protein

Weisfeiler-Lehmann kernel

What does the implementation look like?

```
def compute_prehashed_kernel_matrix(
    self, X: Iterable, Y: Union[Iterable, None]
) -> Iterable:
    def parallel_dot_product(lst: Iterable) -> Iterable:
        res = list()
        for x in lst:
            res.append(dot_product(x))
        return res

    def dot_product(dict1: dict, dict2: dict) -> int:
        running_sum = 0
        # 0 * x = 0 so we only need to iterate over common keys
        for key in set(dict1.keys()).intersection(dict2.keys()):
            running_sum += dict1[key] * dict2[key]
        return running_sum

    if Y == None:
        Y = X

    # It's faster to process n_jobs lists than to have one list and
    # dispatch one item at a time.
    iters = list(chunks(list(itertools.product(X, Y)), self.n_jobs))

    return flatten_lists(
        distribute_function(
            parallel_dot_product,
            iters,
            "Dot product of elements in matrix",
            n_jobs=self.n_jobs,
        )
    )
```

Figure 2: Computing the dot product of the feature maps in parallel.

Weisfeiler-Lehmann kernel

How does it perform?

```
(proteingnnmetrics)
~/Documents/Git/msc_thesis/exploring/kernel_exploration on main 11:16:39
$ python kernel_matrix_computations.py
python kernel_matrix_computations.py
Data path: /Users/philiphartout/Documents/Git/msc_thesis/data
Loading proteins from /Users/philiphartout/Documents/Git/msc_thesis/data/.cache/sample_human_proteome_alpha_fold
### Grakel Implementation ###
Function Name      :compute_naive_kernel
Time               :24.521407292 seconds
Function Name      :compute_naive_kernel
Current memory usage:0.419422MB
Peak              :202.385493MB
### Custom Implementation *without* precomputed W-L hashes ###
Computing Weisfeiler-Lehman Hashes: 100%|
Dot product of elements in matrix: 100%| 100/100 [00:05<00:00, 17.78it/s]
Function Name      :compute_hashes_then_kernel
Time               :7.004050707999999 seconds
Function Name      :compute_hashes_then_kernel
Current memory usage:434.53531MB
Peak              :478.530507MB
### Custom Implementation *with* precomputed W-L hashes ###
Dot product of elements in matrix: 100%| 6/6 [00:01<00:00, 4.44it/s]
Function Name      :compute_kernel_using_precomputed_hashes
Time               :1.3574902499999999 seconds
Function Name      :compute_kernel_using_precomputed_hashes
Current memory usage:0.300306MB
Peak              :44.184051MB
```

Figure 3: Performance and memory footprint of grakel vs. custom. Both are done with 10 iterations of the W-L hashing step.

