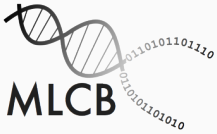
Kernels for proteins and MMD.

Philip Hartout

March 3, 2022



D BSSE

ETH zürich

Introduction

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

Linear kernel

- Not much to optimize there.

Weisfeiler-Lehmann kernel

- Computing $\phi(G)$ needs to be done explicitely 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?

```python
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?

```python
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(dicts: Tuple) -> int:
        running_sum = 0
        # 0 * x = 0 so we only need to iterate over common keys
        for key in set(dicts[0].keys()).intersection(dicts[1].keys()):
            running_sum += dicts[0][key] * dicts[1][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?

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

References i