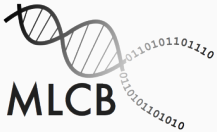
Progress update

Philip Hartout

March 3, 2022



D BSSE

ETH zürich

Introduction

- W-L implementation
- Go through MMD code

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.

MMD

MMD implementations are different, why is the estimate more useful?

```python
def mmd(X, Y, kernel, estimate_variance=False):
    """Calculate MMD between two sets of samples, using a kernel.
    """
    X = np.asarray(X)
    Y = np.asarray(Y)

    # Following the original notation of the paper
    m = X.shape[0]
    n = Y.shape[0]

    K_XX = kernel(X, X)
    K_YY = kernel(Y, Y)
    K_XY = kernel(X, Y)

    # We could also skip diagonal elements in the calculation above but
    # this is more computationally efficient.
    np.fill_diagonal(K_XX, 0)
    np.fill_diagonal(K_YY, 0)

    K_XX = np.sum(K_XX)
    K_YY = np.sum(K_YY)
    K_XY = np.sum(K_XY)

    mmd = 1 / (m * (m - 1)) * K_XX \
        + 1 / (n * (n - 1)) * K_YY \
        - 2 / (m * n) * K_XY

    if estimate_variance:
        var = mmd_variance_estimate(K_XX, K_YY, K_XY)
        return mmd, var

    return mmd
```

Figure 4: MMD estimate, from ICLR `graphgeneval`

```python
class MaximumMeanDiscrepancy(DistanceFunction):
    """Implements maximum mean discrepancy"""

    def __init__(self, kernel: Kernel):
        self.kernel = kernel

    def evaluate(self, X: Any, Y: Any) -> float:
        Xt = check_dist(X)
        Yt = check_dist(Y)

        # Following the original notation of the paper
        m = len(Xt)
        n = len(Yt)

        K_XX = self.kernel.transform(Xt)
        K_YY = self.kernel.transform(Yt)
        K_XY = self.kernel.transform(Xt, Yt)

        # We could also skip diagonal elements in the calculation above but
        # this is more computationally efficient.

        K_XX = np.sum(K_XX)
        K_YY = np.sum(K_YY)
        K_XY = np.sum(K_XY)

        mmd = 1 / (m ** 2) * k_XX + 1 / (n ** 2) * k_YY - 2 / (m * n) * k_XY

        return math.sqrt(mmd)
```

Figure 5: MMD computation, from `proteinggnnmetrics`

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unproces
data that should have been added to the final page this extra page has been added to receiv

If you rerun the document (without altering it) this surplus page will go away, because LaTeX
now knows how many pages to expect for this document.