Progress update

Philip Hartout

March 4, 2022

Introduction

- W-L implementation
- Go through MMD code
- Discuss distances with TDA representations
- Some biological considerations for later

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?

```
(proteinggnnmetrics)
~/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%|████████████████████████████████████████| 100/100 [00:05<00:00, 17.78it/s]
Dot product of elements in matrix: 100%|█████████████████████████████████████████| 6/6 [00:01<00:00,  4.39it/s]
Function Name       :compute_hashes_then_kernel
Time                :7.004050707999998 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                :1.357498249999999 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.

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`

MMD

```python
def main():
    pdb_files = list_pdb_files(HUMAN_PROTEOME)

    if REDUCE_DATA:
        pdb_files = random.sample(pdb_files, 100)

    half = int(len(pdb_files) / 2)

    feature_pipeline = [
        ("coordinates", Coordinates(granularity="CA", n_jobs=N_JOBS),),
        ("contact map", ContactMap(metric="euclidean", n_jobs=N_JOBS)),
        ("knn graph", KNNGraph(n_neighbors=4, n_jobs=N_JOBS)),
        (
            "degree histogram",
            DegreeHistogram("knn_graph", n_bins=30, n_jobs=N_JOBS),
        ),
    ]

    feature_pipeline = pipeline.Pipeline(feature_pipeline, verbose=100)

    dist_1 = feature_pipeline.fit_transform(pdb_files[half:])
    dist_2 = feature_pipeline.fit_transform(pdb_files[:half])

    mmd = MaximumMeanDiscrepancy(
        kernel=LinearKernel(dense_output=False)
    ).fit_transform(dist_1, dist_2)
    print(f"MMD computed from pipeline is {mmd}")
```

Figure 6: Following `sklearn` standards

Kernels for TDA features

The argument for using TDA features for MMD is that the (kernelized) descriptor functions needs to be "rich enough". TDA features are very expressive and operate on the contact map, a rich representation of any protein.

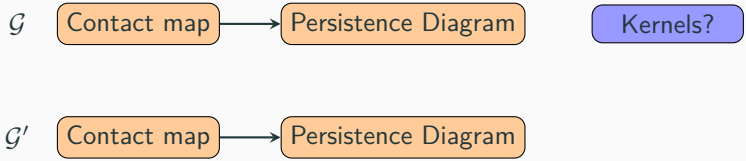

Figure 7: kernels on TDA features.

Ideas: computer pairwise distances. Suppose persistence diagrams $\mathcal{P}_0, \mathcal{P}_1 \sim \mathcal{G}$ and $\mathcal{P}_2, \mathcal{P}_3 \sim \mathcal{G}'$. Can we do: $K_W(\mathcal{G}, \mathcal{G}') = \begin{bmatrix} W_p(\mathcal{P}_0\mathcal{P}_2) & W_p(\mathcal{P}_1\mathcal{P}_2) \\ W_p(\mathcal{P}_0\mathcal{P}_3) & W_p(\mathcal{P}_1\mathcal{P}_3) \end{bmatrix}$?

Where $W_p(\mathcal{P}, \mathcal{P}')$ is the p-Wasserstein distance between persistence diagrams.

The Wasserstein distance *is also a metric*. Some work by [1] suggest this is more complex.

Motifs

Measure *expressivity* of metric by comparing looking at proteins exhibiting structural motifs as a significant part of their structure. Example of motifs:

- Beta-hairpins
- Greek key
- Omega loop
- Helix-loop-helix
- Zinc fingers
- Helix-turn-helix
- Nest/niche

References i

J. H. Oh, M. Pouryahya, A. Iyer, A. P. Apte, A. Tannenbaum, and J. O. Deasy.
Kernel wasserstein distance.
arXiv preprint arXiv:1905.09314, 2019.