STA663 Final Report: Implementation and Optimization of Sinkhorn Algorithm

Github link: https://github.com/congwei-yang/663-Final-Project (https://github.com/congwei-yang/663-Final-Project)

Authors: Congwei Yang, Yijia Zhang, Haoliang Zheng

Contribution

Congwei Yang: writer and programmer. Abstract, Introduction, Conclusion.

Yijia Zhang: writer and programmer. Optimization, Comparison, Upload.

Haoliang Zheng: checker and coordinator. Real Data, Examples, Tests.

Installation

Use the following command in the terminal to install the package:

```
pip install --index-url https://test.pypi.org/simple/ sinkhorn_663
```

Note that the package requires the up-to-date pybind11(2.6.2). Please install or update the pybind11 before installing the sinkhorn package.

To import all the modules and functions, use the following code:

```
from sinkhorn_663 import sinkhorn, log_domain_sinkhorn, sinkhorn_numba, sinkhorn_numba_parallel

from sinkhorn_663 import sample_to_prob_vec, sample_to_prob_vec_nD

from sinkhorn_663.image import cost_mat, flatten, remove_zeros

from skh_cpp import sinkhorn_cpp
```

For detailed documentation of functions, see the github repo above.

Abstract

This paper describes the implementations of the Sinkhorn algorithm [1] proposed by M. Cuturi in 2013. The algorithm provided an efficient approximation to the optimal transport (OT) distance. We built the sinkhorn implementation package `sinkhorn_663` and incorporated numba and c++ to optimize the Sinkhorn function. We also provided a few additional module for the user to conveniently convert random samples or images into empirical measures for Sinkhorn computation. Furthermore, we conducted experiments and tests to explore the properties of Sinkhorn algorithm. Primarily, we tested Sinkhorn algorithm for image testing and compared the Sinkhorn algorithm with the classical linear programming method proposed in [2]. Moreover, we addressed the numerical instability issue of Sinkhorn and compared it with the log-domain sinkhorn proposed in [5] as a pre-existed solution of the issue. Finally, we explored the trend of iteration number required in Sinkhorn algorithm with respect to the data size and regularization parameter λ, and concluded that the a large

regularization parameter λ will cause the algorithm requiring a much larger iteration number. Overall, the Sinkhorn algorithm provides good approximation to the optimal transport distance while maintains relatively low computational cost. However, it is not perfect and has its own disadvantages. Thus, when solving actual problems, we should choose the most appropriate method based on the scenario to maintain sufficient accuracy and efficiency.

Keywords:

optimal transport, Sinkhorn distance, optimization, MNIST data set

1. Background

The optimal transport distance has crucial applications in the field of data anslysis and machine learning. Theoretically, optimal transport distance is a metric and can quantify the distance between probability measures with specified cost function. However, despite its potential applications and favorable properties, optimal transport is generally infeasiable in most problems because of its high computational cost and the curse of dimensionality. In 2013, The Sinkhorn algorithm [1] was proposed by Marco Cuturi. It encorporates entropy regularization to reduce the computational burden, thus provided a much more efficient alternative of the original optimal transport distance. However, the entropy regularization also introduced bias to the framework, causing the Sinkhorn evaluated results to be strictly larger than the exact optimal transport distance.

In this project, we will build a python package to compute the Sinkhorn distance. Moreover, the package provides functions that can convert random samples from probability measures into corresponding format for the computation of 1-Wasserstein distance, which is a special case of optimal transport distance.

Definitions

Before introducing the algorithm, we need to present definitions of the optimal transport problem and Wasserstein distance.

Definition 1: (The Monge-Kantorovich Optimal Transport Problem) Denote the Borel probability measure of $\mathbb{R}^d$ as $\mathcal{P}(\mathbb{R}^d)$. Given a cost function $c(x, y)$, the optimal transport problem between probability measures $P, Q \in \mathcal{P}(\mathbb{R}^d)$ is defined as

$$d(P, Q) := \min_{\pi \in \Gamma(P,Q)} \int_{\mathbb{R}^d \times \mathbb{R}^d} c(x, y) d\pi(x, y)$$

where π is the coupling of P and Q, (i.e. $(x, y) \sim \pi \to x \sim P, y \sim Q$).

We can define the p-Wasserstein distance from a special case of optimal transport with $c(x, y) = \|X - Y\|^p$

Definition 2: From [3], with $p \in \mathbb{N}^+$ and probability measures $P, Q \in \mathcal{P}(\mathbb{R}^d)$ with finite p-th moment, the p-th Wasserstein Distance is defined as

$$W_p(P, Q) := \left(\inf_{\pi \in \Gamma(P,Q)} \int_{\mathbb{R}^d \times \mathbb{R}^d} \|X - Y\|^p d\pi \right)^{1/p}$$

Since it is generally infeasible to directly evaluate probability measures, we need to approximate them using empirical measures. Thus, we need to further specify the optimal transport problem for empirical measures.

Definition 3: From [1], let $r, c \in \{x \in \mathbb{R}_+^d, x^T \mathbf{1}_d = 1\}$ be empirical probability measures. We can define

$$U(r,c) := \{P \in \mathbb{R}_+^{d\times d} : P\mathbf{1}_d = r, P^T\mathbf{1}_d = c\}$$

Then, the optimal transport distance between r and c given a cost matrix M is defined as

$$d_M(r,c) := \min_{P\in U(r,c)} \langle P, M\rangle$$

The cost matrix M is a metric matrix. In other word, M belongs to the cone of distance matrices:

$$\{M \in \mathbb{R}_+^{d\times d} : \forall i,j \le d, m_{ij} = 0 \leftrightarrow i = j, \forall i,j,k \le d, m_{ij} \le m_{ik} + m_{kj}\}$$

With the definition of optimal transport problem for empirical measures, we can proceed to state the definition of Sinkhorn distance.

Definition 4: From [1], denote the entropy of r, c, P to be $h(r), h(c), h(P)$ respectively, then we can introduce the convex set

$$U_\alpha(r,c) = \{P \in U(r,c) \mid h(P) \ge h(r) + h(c) - \alpha\}$$

Then, we can define the Sinkhorn distance by

$$d_{M,\alpha}(r,c) := \min_{P\in U_\alpha(r,c)} \langle P, M\rangle$$

In practice, we will consider a Lagrange multiplier for the entropy constraint of the Sinkhorn distance [1]. For $\lambda > 0$,

$$d_M^\lambda(r,c) := \langle P^\lambda, M\rangle$$

where $P^\lambda = argmin_{P\in U(r,c)} \langle P, M\rangle - \frac{1}{\lambda}h(P)$ [1]. By duality theory we know that each α corresponds a $\lambda > 0$ such that $d_{M,\alpha}(r,c) = d_M^\lambda(r,c)$ holds for a fixed pair of r and c [1].

2. Description of Algorithm

Our package implements the Sinkhorn algorithm proposed by [1] that computes d_M^λ for a specified pair of empirical measures r and c. The algorithm can be viewed as a matrix scaling, since it has been known in transport theory that the solution P^λ has the form of $P^\lambda = \mathbf{diag}(u)K\mathbf{diag}(v)$, where u and v are two non-negative vectors, and $K := e^{-\lambda M}$, the elementary exponential of the matrix $-\lambda M$ [1]. Hence, matrix P^λ can be calculated by Sinkhorn's fixed point iteration $(u,v) \leftarrow (r./Kv, c./K'u)$. Furthermore, the iteration could been simplified to one single iteration $u \leftarrow 1./(\tilde{K}(c./K'u))$. Thus, the algorithm can be easily implemented by iterations of matrix operations as shown in the following figure from [1]. Note that we can use the same steps when C is a matrix whose columns are several vectors $c_1, c_2, \ldots, c_N$ and calculate multiple distances at once.

Algorithm 1 Computation of $\mathbf{d} = [d_M^\lambda(r,c_1), \cdots, d_M^\lambda(r,c_N)]$, using Matlab syntax.

Input $M, \lambda, r, C := [c_1, \cdots, c_N]$.
$I = (r > 0); r = r(I); M = M(I,:); K = \exp(-\lambda M)$
$u = \texttt{ones}(\texttt{length}(r), N)/\texttt{length}(r);$
$\tilde{K} = \texttt{bsxfun}(\texttt{@rdivide}, K, r)$ % equivalent to $\tilde{K} = \mathbf{diag}(1./r)K$
while u changes or any other relevant stopping criterion **do**
  $u = 1./(\tilde{K}(C./(K'u)))$
end while
$v = C./(K'u)$
$\mathbf{d} = \texttt{sum}(u. * ((K. * M)v)$

The entropy regularization will introduce bias to the result of Sinkhorn algorithm, causing the result to be strictly larger than the exact optimal transport distance. Thus, for a probability measure P, we will have $d_\lambda(P, P) > 0$, which means the Sinkhorn distance is no longer a metric.

3. Implementation of the Algorithm

3.1 Basic Implementation

In [1]:

```
from sinkhorn_663 import sinkhorn, log_domain_sinkhorn, sinkhorn_numba, sinkhorn_num
from sinkhorn_663 import sample_to_prob_vec, sample_to_prob_vec_nD
from skh_cpp import sinkhorn_cpp
from sinkhorn_663.image import cost_mat, flatten, remove_zeros
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

First, we follow the above description of algorithm to implement it in plain Python as a baseline. Since the basic iteration steps $u \leftarrow 1./(\tilde{K}(c./K'u))$ have already been established using matrixs and vectors, we utilize package *numpy* to do matrix operations. We use $|u_{new} - u| < tol$ as the stopping criterion.

In [2]:

```
def sinkhorn_plain(r, C, M, lamda, tol = 1e-6, maxiter = 10000):
    M = M[r > 0]
    r = r[r > 0]
    K = np.exp(-lamda * M)
    N = np.shape(C)[1]
    u = np.ones((len(r), N)) / len(r)
    K_tilde = np.diag(1/r) @ K
    d_prev = np.repeat(2., N)
    d = np.ones(N) + 0.5
    for i in range(maxiter):
        u_new = 1/(K_tilde @ (C / (K.T @ u)))
        if np.max(np.abs(u_new - u)) <= tol:
            break
        u = u_new
    v = C/(K.T @ u)
    d = np.sum(u * ((K * M) @ v), axis = 0)
    return d, i
```

3.2 Sanity Check with Simulated Data

To verify the correctness of our basic implementation, we generate some simulation data with known truth and test the algorithm on it. Further optimized algorithms would also be tested on them to check results and compare performance.

As a start, we consider the situation where the empirical measures r and c come from the same distribution and should have a distance close to zero. We generate two groups of samples from a same distribution $\mathrm{Beta}(1, 2)$, use the function *sample_to_prob_vec* to convert samples to vectors and cost matrix as inputs, and

calculate the distance by our function. Here we choose sample size $N = 3000$, $\text{maxiter} = 10000$, $\text{tol} = 1e-6$, $\lambda = 20$. The result is close to 0 es expected. It is a little larger than 0 because of the entropy regularization.

In [3]:

```
# create simulation data
N = 3000
np.random.seed(1)
u1 = np.random.beta(a = 1, b = 2, size = N)
v1 = np.random.beta(a = 1, b = 2, size = N)
M1, r1, c1 = sample_to_prob_vec(u1, v1)
c1 = c1.reshape(-1, 1)
# set parameters
maxiter = 10000
tol = 1e-6
lamda = 20
```

In [4]:

```
sinkhorn_plain(r1, c1, M1, lamda, tol, maxiter)
```

Out[4]:

```
(array([0.04785142]), 36)
```

Then we test on another simulation data from distributions with a setted known OT distance. Distributions $\text{Uniform}(0, 1)$ and $\text{Uniform}(10, 11)$ are used, which have a known OT distance $= 10$. The output of *Sinkhorn_plain* is close to 10 and a little larger than 10 as expected.

In [5]:

```
np.random.seed(1)
u2 = np.random.uniform(0, 1, size = N)
v2 = np.random.uniform(10, 11, size = N)
M2, r2, c2 = sample_to_prob_vec(u2, v2, 1)
c2 = c2.reshape(-1, 1)
```

In [6]:

```
sinkhorn_plain(r2, c2, M2, lamda, tol, maxiter)
```

Out[6]:

```
(array([10.06164666]), 369)
```

3.3 Profiling

Before diving deeper into optimization, we first use `%lprun` in line_profiler to profile the plain version of implementation. Due to the fact that we are operating on matrices and vectors, the plain Python function `Sinkhorn_plain()` is already vectorized when computing. However, the loop structure is still needed because of the sequential updating.

In [7]:

```
%load_ext line_profiler
```

In [8]:

```
%lprun -T plain_profile -f sinkhorn_plain sinkhorn_plain(r2, c2, M2, lamda, tol, max
print(open('plain_profile', 'r').read())
```

```
*** Profile printout saved to text file 'plain_profile'.
Timer unit: 1e-06 s

Total time: 3.92466 s
File: <ipython-input-2-744322bef8cd>
Function: sinkhorn_plain at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def sinkhorn_plain(r,
C, M, lamda, tol = 1e-6, maxiter = 10000):
     2         1      14161.0  14161.0      0.4      M = M[r > 0]
     3         1         76.0     76.0      0.0      r = r[r > 0]
     4         1      79259.0  79259.0      2.0      K = np.exp(-lamda
* M)
     5         1         15.0     15.0      0.0      N = np.shape(C)
[1]
     6         1         34.0     34.0      0.0      u = np.ones((len
(r), N)) / len(r)
     7         1     207706.0 207706.0      5.3      K_tilde = np.diag
(1/r) @ K
     8         1         77.0     77.0      0.0      d_prev = np.repea
t(2., N)
     9         1         21.0     21.0      0.0      d = np.ones(N) +
0.5
    10       370       1013.0      2.7      0.0      for i in range(ma
xiter):
    11       370    3525072.0   9527.2     89.8          u_new = 1/(K_
tilde @ (C / (K.T @ u)))
    12       370      56109.0    151.6      1.4          if np.max(np.
abs(u_new - u)) <= tol:
    13         1          4.0      4.0      0.0              break
    14       369       2153.0      5.8      0.1          u = u_new
    15         1       4512.0   4512.0      0.1      v = C/(K.T @ u)
    16         1      34450.0  34450.0      0.9      d = np.sum(u *
((K * M) @ v), axis = 0)
    17         1          2.0      2.0      0.0      return d, i
```

According to the profile result, we can see most of the time is cost on computing matrix multiplication. The step `u_new = 1/(K_tilde @ (C / (K.T @ u)))` costs most of the total time.

To further improve the performance, we decide to use:

1. JIT compilation and parallelism with numba
2. Re-writing critical functions in C++ and using pybind11 to wrap them

4. Optimization

4.1 Methods and Details

To optimize our function, we first utilize JIT compilation with numba, which is the default version in our main function `sinkhorn`. We also develop the parallel version with numba, which can be called by setting `parallel=True` in `sinkhorn`.

Furthermore, we write the function in c++ and use pybind11 to wrap them as `sinkhorn_cpp`. Note that we use **eigen** library to help us do matrix computation. To make our package function well, we include necessary documents of **eigen** in our package directory, which is everything in `sinkhorn_663/Eigen`. Also, before uploading our package, we add a `MANIFEST.in` with `recursive-include sinkhorn_663/Eigen *` to claim that the Eigen directory is included in our package.

In terms of improving the stability, we consider a another algorithm: Sinkhorn in log domain. It has been implemented and complied with numba, and can be called by setting `log_domain=True` in `sinkhorn`. We will talk about why and how we use this algorithm in the later part.

4.2 Improvement of Time

We've used above simulation data to verify that relevant sinkhorn functions are correct. See test codes in `tests/test_sinkhorn_functions.py`.

Then we compare the time to see the improvement of speed after optimization. Numba and numba with parallel is comparative to the plain version in speed. The reason may be that the functions already use vector and matrixs. C++ version is much faster than previous ones. For faster speed, we recommend to use `sinkhorn_cpp` as our final optimized implementation.

In [9]:

```
%timeit sinkhorn_plain(r2, c2, M2, lamda, tol, maxiter)
%timeit sinkhorn(r2, c2, M2, lamda, tol, maxiter)
%timeit sinkhorn(r2, c2, M2, lamda, tol, maxiter, parallel = True)
%timeit sinkhorn_cpp(r2, c2, M2, lamda, tol, maxiter)
```

```
3.6 s ± 161 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
3.51 s ± 31.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
2.59 s ± 214 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
395 ms ± 20.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

5. Tests and Examples

5.1 Other Tests

We have discussed how we did sanity check for sinkhorn functions. Except different versions of sinkhorn functions, we also include other functions which help us handle the real and simulated data. For example, we have used `sample_to_prob_vec()` in the earlier part. We will use functions like `cost_mat()`, `flatten()` in the later part. We create several other tests to make sure all functions are performing well. All the tests pass. They are:

1) Test the `sample_to_prob_vec()` returns the right dimensions and probability vector sum up to 1.

2) Test the `cost_mat()` returns the right dimensions.

3) Test the `flatten()` returns the right length of list and each vector in the list sum up to 1.

5.2 Examples

In `examples/` directory, we present codes showing how to use each function, real data set and repeat our results. For functions, we show the inputs and outputs. For data, because they are stored in `.mat` format, we show how to read in and extract the information. For results, all our following results can be reproduced by codes in this directory.

1) `example_sample_to_prob_vec.py` and `example_sinkhorn_functions.py` are codes showing how to use corresponding functions.

2) `example_compare_EMD.py` , `example_complexity.py` `example_numerical_instability.py` , and `example_data_silhouettes.py` are codes using data and producing results which we will talk about in the following parts.

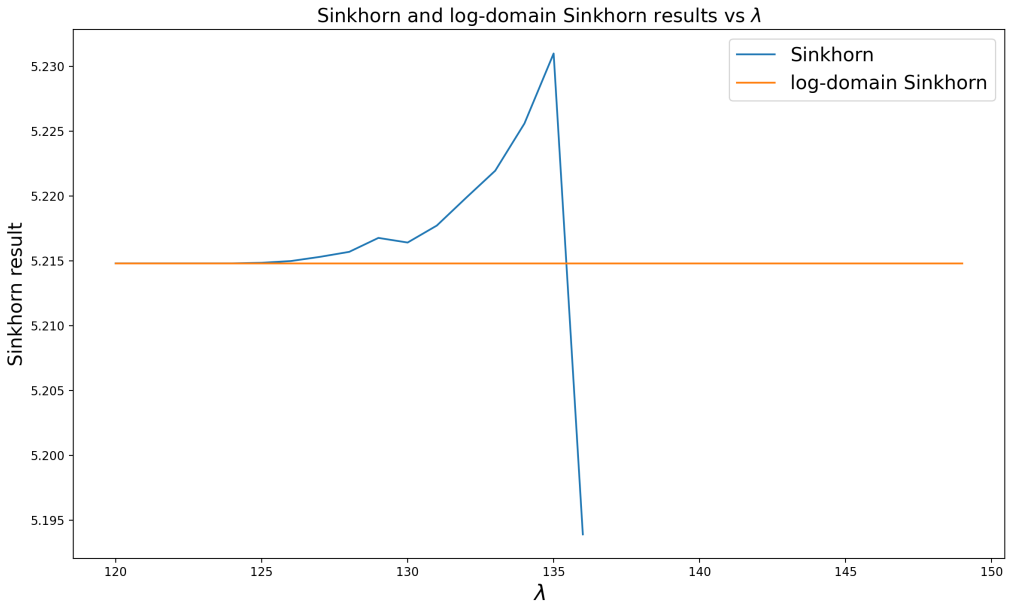
6. Comparative Analysis

6.1 Comparison with Sinkhorn Algorithm in log domain: Numerical Instability

The Sinkhorn algorihtm brought a great leap in computational efficiency of optimal transport distance and relieved the curse of dimensionality. However, it suffers from numerical instability [4]. Notice that when λ has a large scale, the matrix $K = e^{-\lambda M}$ may have extremely small terms, causing the following matrix scaling process to produce extremely large or small values [4]. This will lead to inaccurate numerical outputs, and possibly overflow or underflow problem in the algorithm [4].

Fortunatley, a solution to this numerical instability problem is to perform the matrix scaling in log-domain [5], where the numerical values in the computation process can be stabilized in an acceptable range. Indeed, the log-domain sinkhorn supports a wider range of λ. However, it also has disadvantages. The log-domain Sinkhorn incorporates much more exponential and logarithm operations, and it is not longer in the form of simple matrix-vector operations. Thus, the log-domain Sinkhorn suffers from heavier computational cost, and it can not be easily parallelized for multiple pairs of empirical measures. In practice, we can try to avoid the λ range that will cause numerical instability and use the original Sinkhorn algorithm as much as possible, and only turn to the log-domain Sinkhorn when necessary.

To compare the numerical stability of Sinkhorn and log domain Sinkhorn, we generate simulation data, choose a large range of λ, and plot the results of them. Log domain Sinkhorn has also been implemented and compiled with numba, and can be called by using `parallel=True` in function `sinkhorn` .



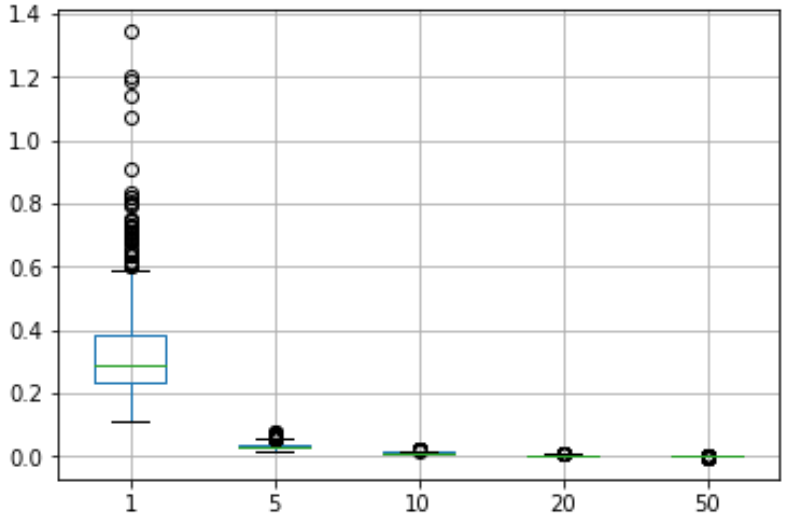
We can see that, with $P : Beta(2, 5)$ and $Q : Uniform(5, 6)$, the Sinkhorn algorithm produces inaccurate results with $\lambda > 125$, and the overflow/underflow occurs at $\lambda = 136$. On the other hand, the log-domain Sinkhorn continue to produce relatively accurate result up to $\lambda = 149$.

Above result was calculated by codes in `exmaples/example_numerical_instability.py` .

6.2 Comparison with EMD on MNIST data set

In this part, we test the algorithm on the MNIST digits data set as a real world example, which has been used in the original paper[1]. Since earth mover's distance (EMD) is the classical optimal transport distance to measure the difference between probility measures, we also want to compare Sinkhorn distance with EMD when applied to real data sets. EMD is typically calculated by the classical linear programming method proposed in [2]. According to the theoretical background that Sinkhorn distance $d_M^\lambda(r, c) = \min_{P \in U(r,c)} \langle P, M \rangle - \frac{1}{\lambda} h(P)$, Sinkhorn distance is expected to be larger than EMD due to the regularization and converge to EMD when $\lambda \to +\infty$. To evaluate this convergence, we choose different λ and plot the distribution of deviation $(d_M^\lambda(r, c) - EMD)/EMD$.

1000 pairs of images are chosen from the MNIST digits data set. Each image would be flattened to a vector of intensities as our input. Cost matrix is the distance between the coordinates of pixels in the image. EMD are calculated using python package *POT* (Python Optimal Transport).



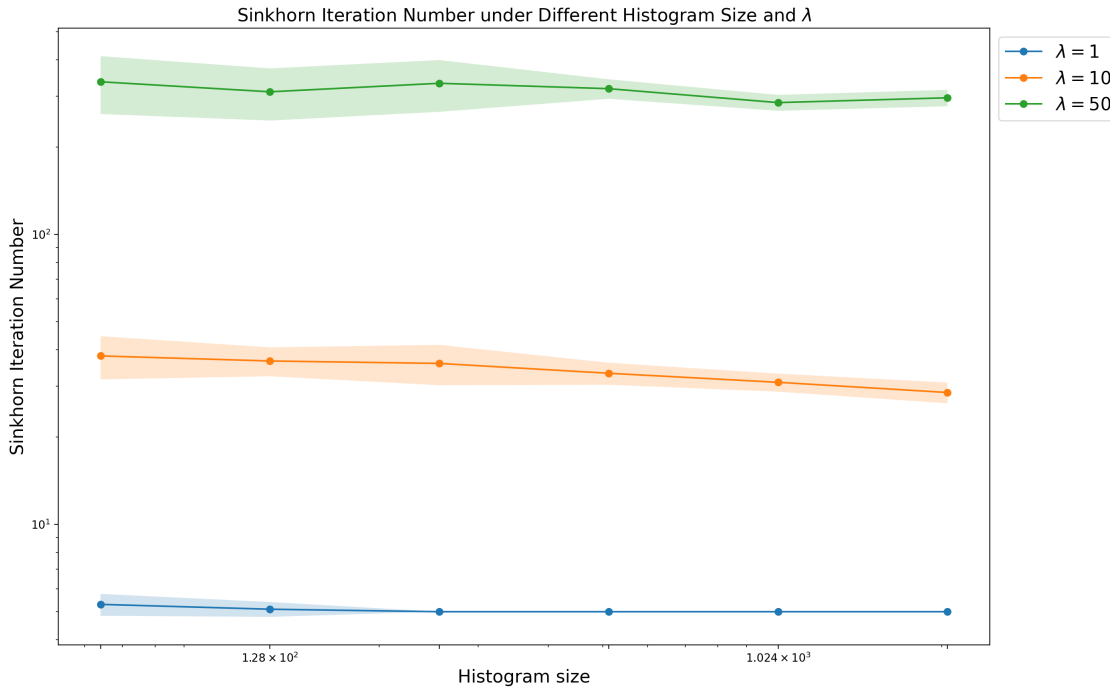
We choose $\lambda \in (1, 5, 10, 20, 50)$. As shown in the boxplot, the deviation $(d_M^\lambda(r, c) - EMD)/EMD$ decreases when we use a larger λ. Especially, when λ is larger than 10, the deviation is close to zero. This is reasonable because when λ is larger, the coefficient of regularization term is smaller, and thus Sinkhorn distance becomes closer to EMD. Hence, if we choose λ from a reasonable range, we could avoid overflow and calculate Sinkhorn distance with a small deviation from EMD at the same time.

Above result was calculated by codes in `exmaples/example_compare_EMD.py` .

7. Applications

7.1 Empirical Complexity on Simulated Data

As a further experiment to explore the property of Sinkhorn algorithms, we generate simulation data with different data sizes and plot the iteration numbers required in Sinkhorn algorithm with respect to the data size and regularization parameter λ.



We tested with a range of data sizes $\in (64, 128, 256, 512, 1024, 2048)$ and $\lambda \in (1, 5, 50)$. For each data size and λ, the experiments are replicated ten times and the same tolerance as convergence criterion is used.

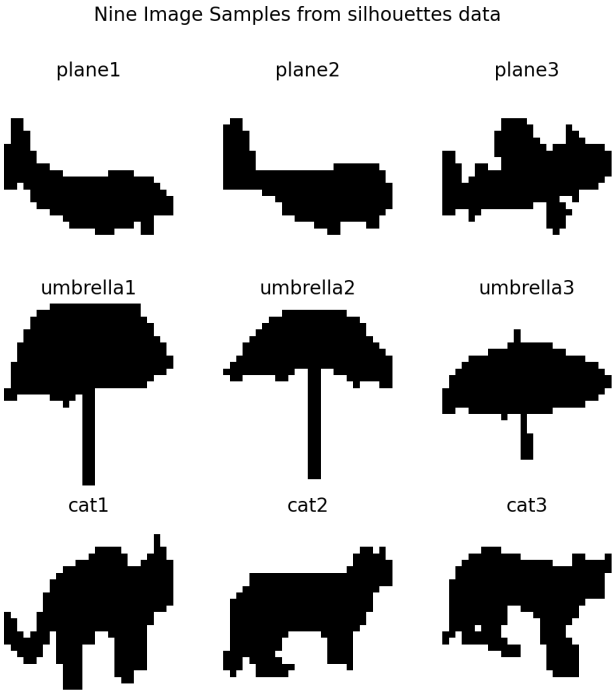
As shown in the plot, when the data size increases, the iteration numbers do not change a lot. When larger λ is chosen, we need a larger iteration number to calculate Sinkhorn distance. These trends correspond to the findings in [1]. We need not set a larger max iteration numbers for larger data sets.

Above result was calculated by codes in `exmaples/example_complexity.py` .

7.2 Applications to Real Data Sets

We've tested on MNIST data set when comparing with EMD. We also found one other real-world data set not in the original paper and tested on it. Because optimal transport distances are computed for theoretical probability measures, it is hard to find real data sets which can directly use this method and describe the result. Therefore, we decide to use another image data set, distances between image matrix can be interpreted as the difference between images.

The image data we choose is [CalTech 101 Silhouettes Data Set (https://people.cs.umass.edu/~marlin/data.shtml)](https://people.cs.umass.edu/~marlin/data.shtml). According to the description of this data, each image in the CalTech 101 data set includes a high-quality polygon outline of the primary object in the scene. The outline is rendered as a filled, black polygon on a white background. Many object classes exhibit silhouettes that have distinctive class-specific features. There are thousands of images contained in this data set. To show a intuitive visual result, we select nine image samples from this data, which are shown below.



There are three objects: plane, umbrella, and cat, and three images for each object. Only through looking at the picture, we may assume that the distance between images for the same object won't be too large. Also, there is "outlier" in each object, plane3, umbrella3 and cat1 to be specific. For these outliers, the distance between other images of the same object may be larger.

The table below shows the result from computing the distance. In all, the result is as expected. First, we can see the values on the diagonal are all close to 0, which is correct. Second, within blocks of each object, the distance is not too large. The largest distance is between umbrella1 and umbrella3. This is reasonable, because areas of these two silhouettes do differ a lot. Third, the largest distance among all entries occur at the distance between different objects.

In [10]:

```
pd.read_pickle("result_df")
```

Out[10]:

	plane1	plane2	plane3	umbrella1	umbrella2	umbrella3	cat1	cat2	cat3
plane1	0.00	0.47	1.65	4.77	3.39	1.84	2.48	2.09	2.23
plane2	0.47	0.00	1.54	4.57	3.15	1.68	2.27	2.09	2.09
plane3	1.65	1.54	0.00	3.77	2.58	0.82	1.43	1.00	1.22
umbrella1	4.77	4.57	3.77	0.00	1.65	3.18	4.03	4.00	3.02
umbrella2	3.39	3.15	2.58	1.65	0.00	1.94	2.89	3.02	2.18
umbrella3	1.84	1.69	0.82	3.18	1.94	0.00	1.85	1.40	0.89
cat1	2.48	2.27	1.43	4.02	2.89	1.85	0.00	1.64	1.92
cat2	2.09	2.09	1.00	4.00	3.02	1.40	1.64	0.00	1.36
cat3	2.23	2.09	1.22	3.02	2.18	0.89	1.92	1.36	0.00

Above dataframe was calculated by codes in `exmaples/example_data_silhouettes.py` .

8. Conclusion

Overall, the Sinkhorn algorithm provides a good approximation to the exact optimal transport distance while maintain relatively low computational cost. As shown in the experiments, with increasing regularization parameter λ, the bias introduced by entropy regularization shrinks to a small scale quickly. However, we have to address that it is not always benificial to use a large λ, since the tests also showed that large regularization parameter λ will lead to a much larger interation number required, thus imposing a heavier computational burden. The best practice would be selecting the smallest λ that can produce results satisfying the accuracy requirement based on the actual application scenario.

On the other hand, the Sinkhorn algorithm also exposed its issues, and the most significant problem is the numerical instability. Even we have log-domain Sinkhorn to resolve it, there is still a price to pay. As discussed previously, the log-domain Sinkhorn loses the simple matrix-vector operation structure, thus can not be easily parallelized. Overall, the Sinkhorn and log-domain Sinkhorn are two algorithms each with advantages and disadvantages, which requires us to apply the appropriate algorithm based on the actual scenario.

Optimal transport distance can be applied in clustering problems. Given a data set, we can use Sinkhorn algorithm to approximate the optimal transport distance, which quantifies the difference between observations, and apply clustering to find out observations with similar features. For image processing, we can use the same workflow to find out similar images.

The improvements to the Sinkhorn algorithm focuses on its two main issues: bias and numerical instability. Since the Sinkhorn algorithm was proposed in 2013 by [1], the field of optimal transport has drawn the attention of a plethora of researchers, and many improvements based on the Sinkhorn algorithm have been proposed. For example, the method "ϵ scaling" is proposed by [4] to resolve the numerical instability issue while maintain the structure of the original algorithm. Moreover, a debiased Sinkhorn algorithm was proposed in [6], and returned the metric property to the Sinkhorn distance.

Reference

[1] M. Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, Advances in Neural Information Processing Systems 26, pages 2292–2300. Curran Associates, Inc., 2013.

[2] Pele, O. and Werman, M. Fast and robust earth mover's distances. In ICCV'09., (2009).

[3] Ramdas, A., Trillos, N. G., and Cuturi, M. On Wasserstein two-sample testing and related families of nonparametric tests. Entropy, 19(2). 2017.

[4] L. Chizat, G. Peyr'e, B. Schmitzer, and F.-X. Vialard, Scaling algorithms for unbalanced optimal transport problems, Math. Comp., 87 (2018), pp. 2563–2609, https://doi.org/10.1090/mcom/3303 (https://doi.org/10.1090/mcom/3303).

[5] M. Cuturi., G. Peyr'e, Computational Optimal Transport. arXiv:1803.00567 [stat.ML]

[6] H. Janati, M. Cuturi, and A. Gramfort, Debiased Sinkhorn Barycenters. arXiv:2006.02575v1 [stat.ML], Jun, 2020