afkmc2 Documentation

Release .1

Adrian Goedeckemeyer

Apr 20, 2017

Contents

CHAPTER 1

Introduction

This package contains sklearn compatible python implementations of various K-Means seeding algorithms.

The package was inspired by the AFKMC^2 algorithm detailed in

> **Fast and Provably Good Seedings for k-Means** *[afkmc2]*
> Olivier Bachem, Mario Lucic, S. Hamed Hassani and Andreas Krause
> In *Neural Information Processing Systems* (NIPS), 2016.
> https://las.inf.ethz.ch/files/bachem16fast.pdf

The algorithm uses Monte Carlo Markov Chain to quickly find good seedings for KMeans and offers a runtime improvement over the common K-Means++ algorithm.

Usage

Using this package to get seedings for KMeans in sklearn is as simple as:

```
import afkmc2
X = np.array([[1, 2], [1, 4], [1, 0],
              [4, 2], [4, 4], [4, 0]])
seeds = afkmc2.afkmc2(X, 2)

from sklearn.custer import KMeans
model = KMeans(n_clusters=2, init=seeds).fit(X)
print model.cluster_centers_
```

Installation

Quickly install afkmc2 by running:

```
pip install afkmc2
```

Contribute

- Issue Tracker: https://github.com/adriangoe/afkmc2/issues
- Source Code: https://github.com/adriangoe/afkmc2

Support

You can reach out to me through https://adriangoe.me/#contact-us.

License

The project is licensed under the MIT License.

CHAPTER 2

Reference

Seeding Reference

View Code On Github.

K-Means++

K-Means++ is the original seeding algorithm for K-Means as proposed by Arthur and Vassilvitskii in 2007 *[kmpp]*.

It chooses one center uniformly, then computes distance of every datapoint to already chosen centers in order to use distance as weights when sampling next center. These steps are repeated until *k* centers are chosen.

Chosing good seedings speeds up convergence for K-Means, but extra time cost is occurred calculating all distances.

`afkmc2.`**`kmpp`**(*X*, *k*)

KMeans++ Seeding as described by Arthur and Vassilvitskii (2007)
Runtime `O(nkd)`

Parameters

- **`X`** (*`np.array`*) – Datapoints. Shape: (n, d)
- **`k`** (*`int`*) – Number cluster centers.

Returns Cluster centers for seeding. Shape: (k, d)

Return type np.array

Example `seeds = afkmc2.kmpp(X, 3)`

K-Means Markov Chain Monte Carlo

KMC^2 was proposed as an improvement over K-Means++ in 2016 *[kmc2]*. While K-Means++ requires k full passes over the dataset, KMC^2 replaces the D^2 sampling step with Markov Chain Monte Carlo sampling. Runtime is no longer tied to number of datapoints while new centers will be chosen far from current centers.

`afkmc2.`**`kmc2`**(*X*, *k*, *m=200*)

KMC^2 Seeding as described by Bachem, Lucic, Hassani and Krause (2016)
Runtime `O(mk^2d)`

Parameters

- **`X`** (*`np.array`*) – Datapoints. Shape: (n, d)
- **`k`** (*`int`*) – Number cluster centers.
- **`m`** (*`int`*) – Length of Markov Chain. Default 200

Returns Cluster centers for seeding. Shape: (k, d)

Return type np.array

Example `seeds = afkmc2.kmc2(X, 3)`

Assumption Free KMC^2

AFKMC^2 is an improvement proposed by the same authors *[afkmc2]*. While KMC^2 requires assumptions about the data generating distribution (in our implementation uniformity), this algorithm works without such assumptions. It the true D^2-sampling distribution with regards to the first center `c_1` as a proposal distribution that can approximate nonuniform distributions.

This means an added runtime cost of `O(nd)` to calculate the initial distribution, but performance improvements for nonuniform samples.

`afkmc2.`**`afkmc2`**(*X*, *k*, *m=200*)

AFKMC^2 Seeding as described by Bachem, Lucic, Hassani and Krause (2016)
Runtime `O(nd + mk^2d)`

Parameters

- **`X`** (*`np.array`*) – Datapoints. Shape: (n, d)
- **`k`** (*`int`*) – Number cluster centers.
- **`m`** (*`int`*) – Length of Markov Chain. Default 200

Returns Cluster centers for seeding. Shape: (k, d)

Return type np.array

Example `seeds = afkmc2.afkmc2(X, 3)`

Cached AFKMC^2

The author of this package proposed this slight runtime improvement for AFKMC^2 (it could also be applied to KMC^2). Since the first `O(nd)` pass already calculates all distances between `X` and `c_1` we can at minimum save

ourselves k*m distance calculations by storing these results to be reused in the Markov Chain. This comes with an additional space cost of `O(nk)`.

The savings are highest for small datasets but can yield significant runtime improvement for very large/high-dimensional ones as well.

`afkmc2.`**`afkmc2_c`**(*X*, *k*, *m=200*)

Cached AFKMC^2 Seeding based on AFKMC
as described by Bachem, Lucic, Hassani and Krause (2016)

Caching addition to prevent duplicate work for small datasets
Additional space cost during execution `O(nk)`
Runtime `O(nd + mk^2d)`

Parameters

- **X** (*np.array*) – Datapoints. Shape: (n, d)
- **k** (*int*) – Number cluster centers.
- **m** (*int*) – Length of Markov Chain. Default 200

Returns Cluster centers for seeding. Shape: (k, d)

Return type np.array

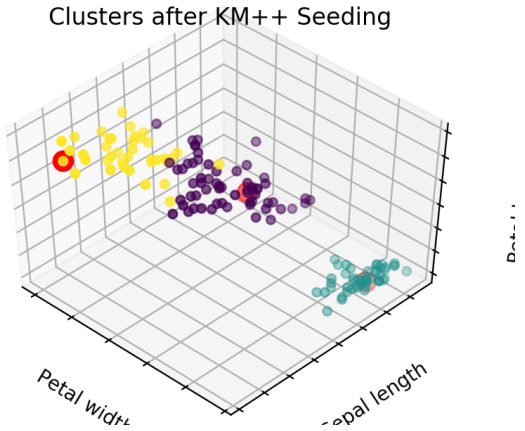
Example `seeds = afkmc2.afkmc2_c(X, 3)`

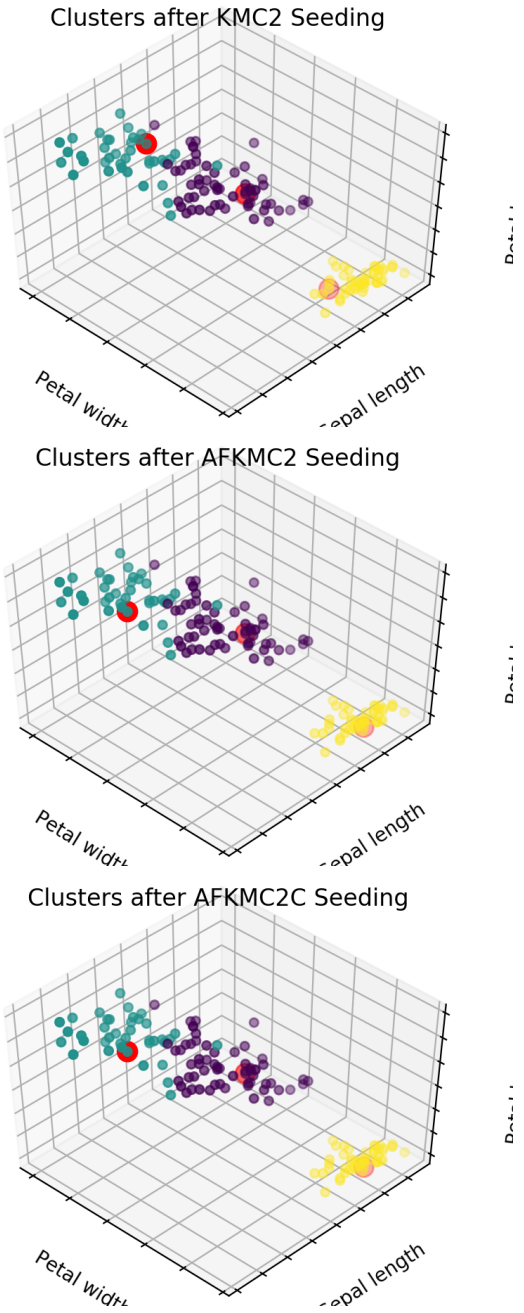
References

Seeding Demonstration

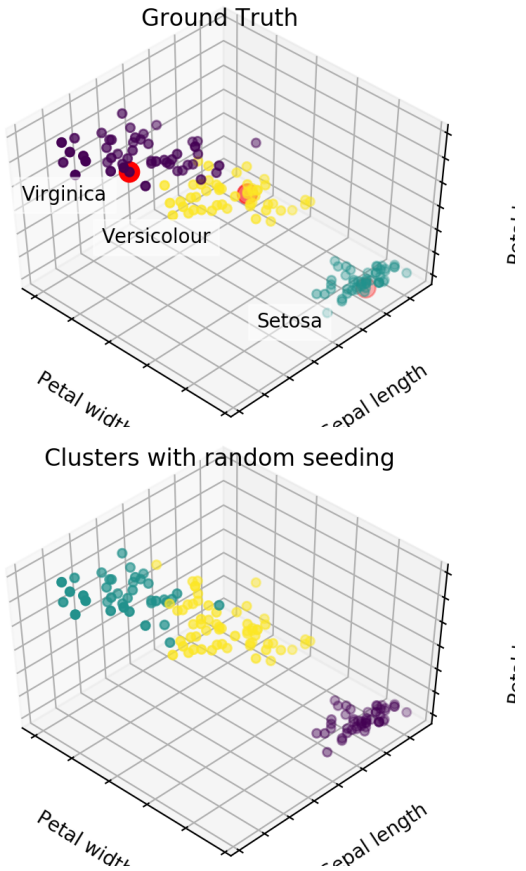
Iris Dataset

An easy way to see the quality of the seedings chosen by each of these algorithms is visualizing the seed choices on top of the frequently used "Iris" dataset in *sklearn*.





We easily see that each algorithm finds reasonable choices for seeds that barely differ from each other. These seedings allow KMeans to categorize the data well, we can see that by comparing tot he ground truth shown below. Clusters with random seedings are shown below as well, we see that KMeans still converges at a good solution since this is an easy problem, but the number of iterations needed to get there was higher.



We expect KM++ and AFKMC2 to have the highest quality seedings while KMC2 might in some cases suffer from a poor choice of assumed distribution. The main difference between KM++ and AFKMC2 will be visible when looking at runtime.

Runtime Comparison

The time complexity of using one of the KMC^2 approaches over KM++ clearly shows for larger datasets.

Average Runtime for 50 passes, 40 dimensions and 3 centers

Size	KM++	KMC2	AFKMC2	AFKMC2C
200	.0031	.0054	.0133	.0107
1000	.014	.0053	.0204	.01899
5000	.07838	.00556	.05683	.058771
20000	.29286	.00529	.17766	.188594
100000	.59260	.0057	.87167	.929336

While on a set with 200 observations and 40 dimensions KM++ outperforms the others, the MCMC approaches bring large time savings for datasets with 2000+ observations. We can still feel the one pass over n in the AF approaches, but if the number of centers increases KM++ would feel a strong increase in runtime while AFKMC2 is barely affected as shown below.

Size	K	Dimensions	KM++	KMC2	AFKMC2	AFKMC2C
100000	3	40	.59260	.0057	.87167	.929336
100000	6	40	6.9294	.01998	1.4054	1.47619
100000	3	80	1.5638	.00559	.86605	.924057
500	20	80	.43924	.20874	.28856	.173561

We notice that the proposed addition of caching reduces performances in situations with numbers of observations. This is due to the fact that we save between `(1-k)*m` and `.5*(1-k)^2*m` passes over the data. Since MCMC does not need to increase m for large datasets we will only save slightly above 1200 calculations for the case with 6 centers and 100000 points but still have to do at least 101000 calculations. Only the last example shows a case in which the time saving due to caching is significant and clearly outperforms all other cases since in a dataset with 500 points we are more likely to have duplicates among our 200 points in the Markov Chain.

Demo Code on GitHub.

Bibliography

[kmpp] Arthur, D., & Vassilvitskii, S. (2007, January). k-means++: The advantages of careful seeding. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (pp. 1027-1035). Society for Industrial and Applied Mathematics.

[kmc2] Bachem, O., Lucic, M., Hassani, S. H., & Krause, A. (2016, February). Approximate K-Means++ in Sublinear Time. In AAAI (pp. 1459-1467).

[afkmc2] Bachem, O., Lucic, M., Hassani, H., & Krause, A. (2016). Fast and Provably Good Seedings for k-Means. In Advances in Neural Information Processing Systems (pp. 55-63).

Index