DARTMOUTH

Randomization & Approximation in Machine Learning and Neural Network Pruning

Math 76 Final Project

Warren Shepard Sri Korandla Tushar Aggarwal

August 2024

Dr. Alice Schwarze

Table of contents

Basic Randomization & Approximation in ML

Randomization & Approximation as we know it

The randomization techniques from class mostly involve **splitting** or **modifying data** randomly:

- splitting data into training and testing subsets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

- bagging/bootstrapping procedures
- random parameter/weight initialization in neural networks

Approximation is of course the goal of machine learning, but we often use *exact* algorithms in ML models instead of approximation algorithms

What else can Randomization & Approximation be used for

- Optimization:
  - trade ”accuracy” for speed
  - oftentimes algorithms have $\geq 90\%$ accuracy
- In neural networks
  - ”Prune” connections (somewhat) at random to get a smaller model with higher accuracy
- Sometimes, allowing some error is more ”realistic” leading to a higher accuracy
  - higher accuracy’s could also be because of a reduction in over fitting

Example: k Approximate Nearest Neighbors

Naive Solution

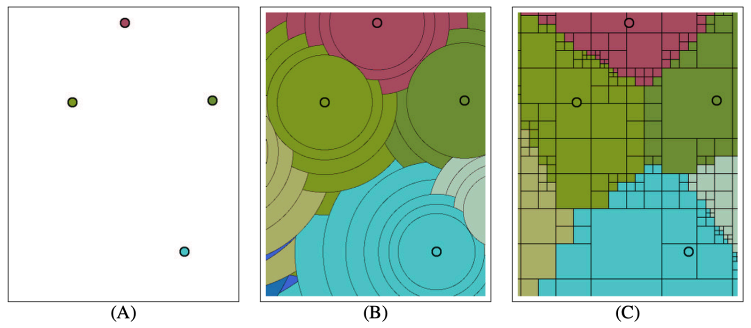
Problem: Given a set of vectors S in a data space and a similarity measure DIST (for example euclidean distance) find the k nearest ("most similar") neighbors to some input x.

Brute force solution: compare to every other point

```
procedure BRUTE-KNN(x, S, DIST, k = 1)
    Compute DIST(x, s) ∀ s ∈ S
    return k points (s ∈ S) with smallest DIST values
```

Issue: expensive and **redundant** (especially in higher dimensions and when there is lots of training data); $O(n)$ lookup

k Approximate Nearest Neighbors (kANN)



A) data points; B) & C) approximate nearest neighbor [3]

Locality Sensitive Hashing (LSH)

Hashing:

- hash function: $f : U \to [n]$ where $|U| >> n$ and n is (usually) prime
- typically, two inputs have a very low probability of ”colliding” (their output is the same)
    - inputs will have different ”hashes” even if they are very similar

Locality Sensitive Hashing (LSH)

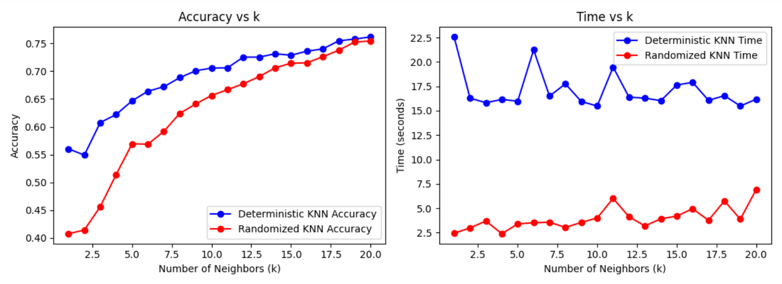
- two inputs which are similar have a high probability of colliding [3]
- can be used to solve kANN with high accuracy and in less time

Approximate-KNN Experiment

- using PyNNdescent package for kANN implementation [5]
  - uses modern kANN algorithm [1], written in python
  - accuracy converges to 90%
- Naive brute force implementation hand written in python
  - did not use an implementation from scikitlearn because compiled in C
  - best to have implementation we are comparing all in the same language

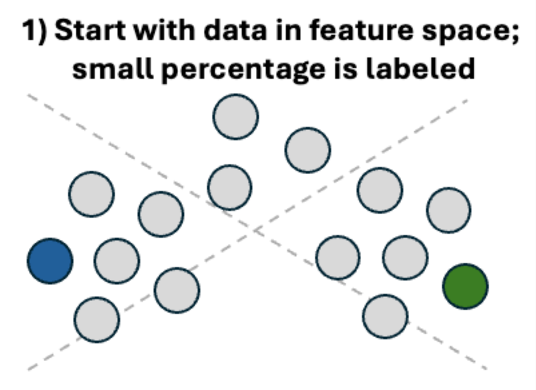
Results

```
# Generate a synthetic dataset with 10,000 samples and 1,000 features
X, y = make_classification(n_samples=10000, n_features=200, n_informative=50, n_classes=3,
       random_state=42)
```

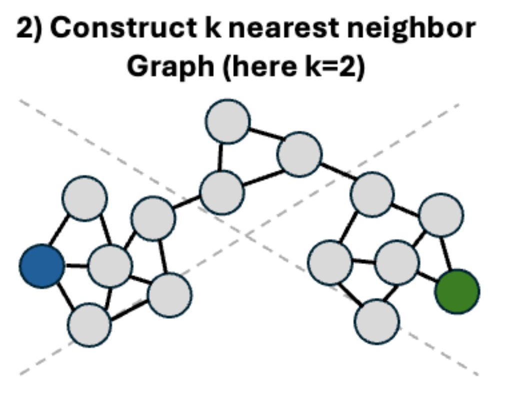


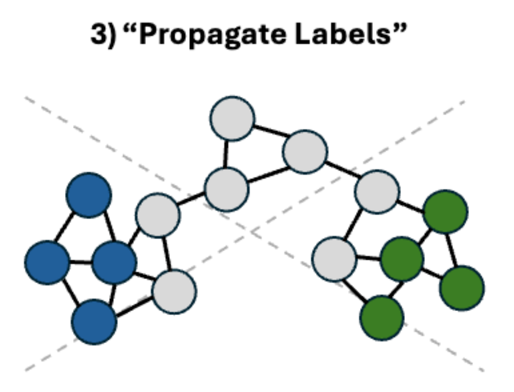
kNN vs kANN **raw** accuracy and time comparison

Graph-Based Semi-Supervised Learning

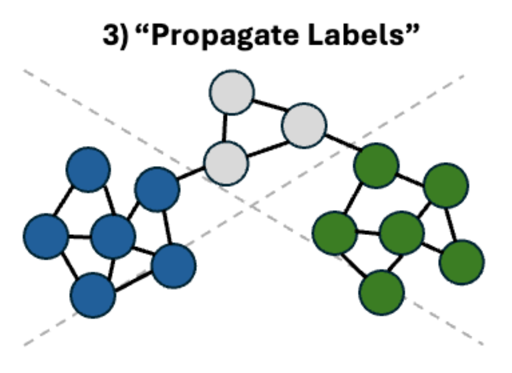

1) Start with data in feature space;
small percentage is labeled

Graph-Based Semi-Supervised Learning

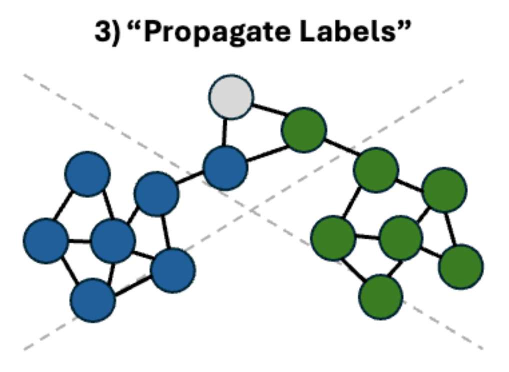



3) "Propagate Labels"

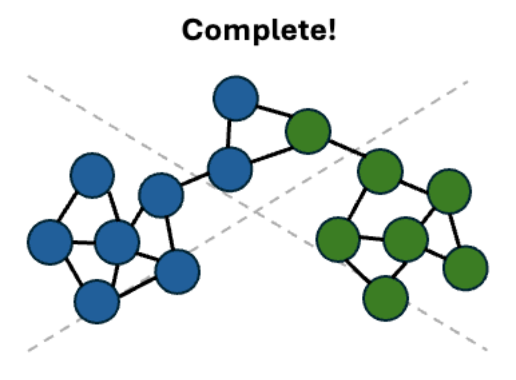
Graph-Based Semi-Supervised Learning



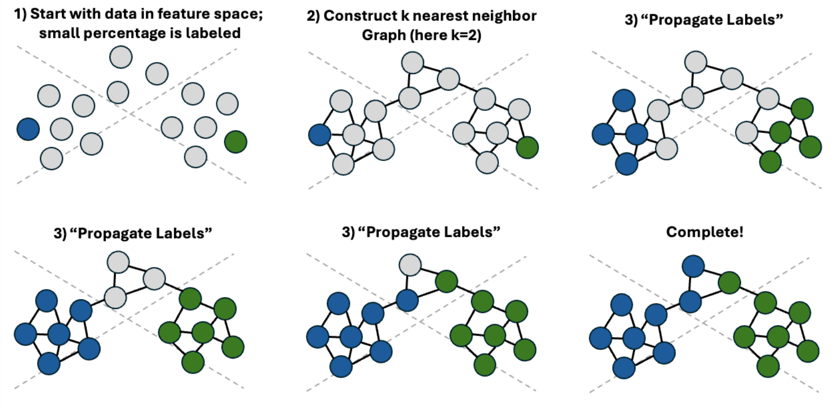
Graph-Based Semi-Supervised Learning



Graph-Based Semi-Supervised Learning



Graph-Based Semi-Supervised Learning



Graph-Based Semi-Supervised Learning

- Constructing the k Nearest Neighbor Graph dominates runtime
- Most Label Propagation Algorithms (LPA) are **simple** and **near linear** in time

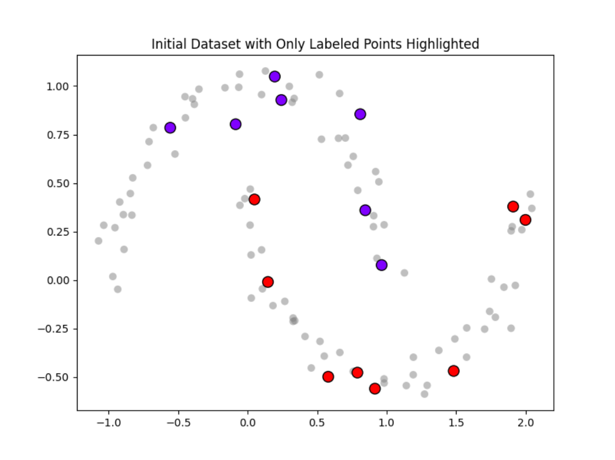
Parameters:

- $y :=$ initial labels; $y_i :=$ label of node i; -1 if unlabeled
- $W :=$ adjacency matrix of nearest neighbors graph
- $\alpha \in [0, 1] :=$ "clamping factor" indicating influence of initial labels **per iteration**

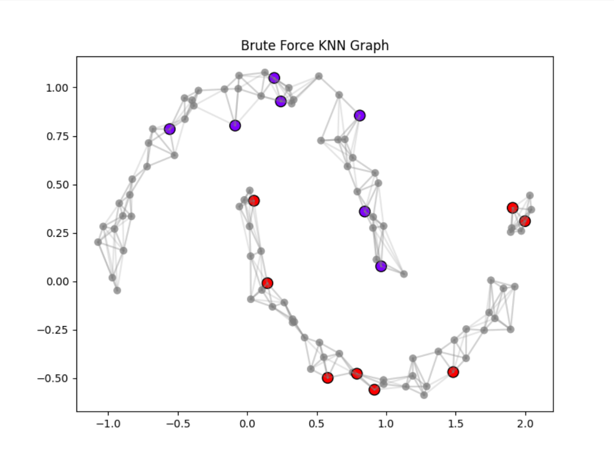
Update Rules:

- $F_{ic}^{(0)} \leftarrow \begin{cases} 1 & \text{if } y_i = c \\ 0 & \text{otherwise} \end{cases}$ $\quad$ F_{ic} is the **confidence** of node i having label c
- $F_{ic}^{(t+1)} \leftarrow \underbrace{\alpha W F^{(t)}}_{\text{propagate from neighbors}} + \underbrace{(1-\alpha) F^{(0)}}_{\text{account for initial labels}}$
- $y_i^* \leftarrow \mathbf{arg\,max}_c\, F_{ic}^{(t_{\max})}$ $\quad$ y_i^* is the vector of **final labels**

Results I

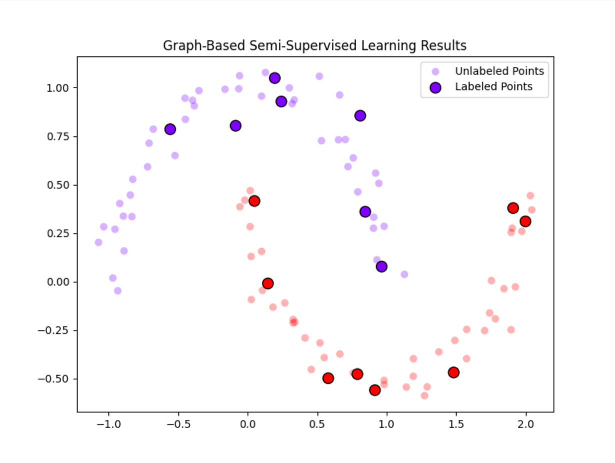

Initial Dataset with Only Labeled Points Highlighted
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

Results I



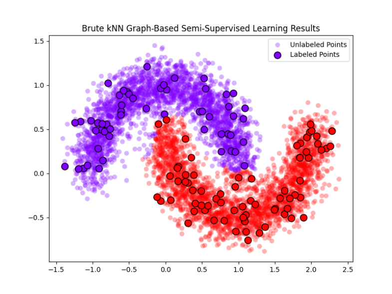
Nearest Neighbor graph with $k = 5$

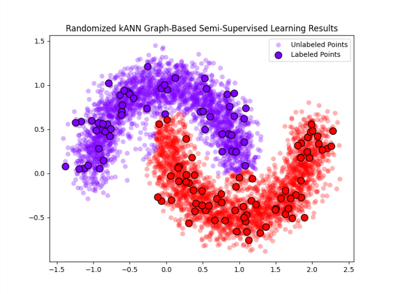
Results I



Final labels using $\alpha = .99$

Results II

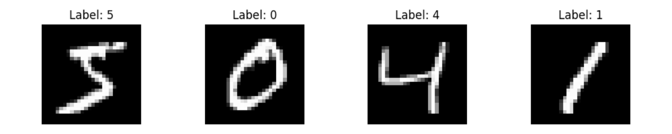




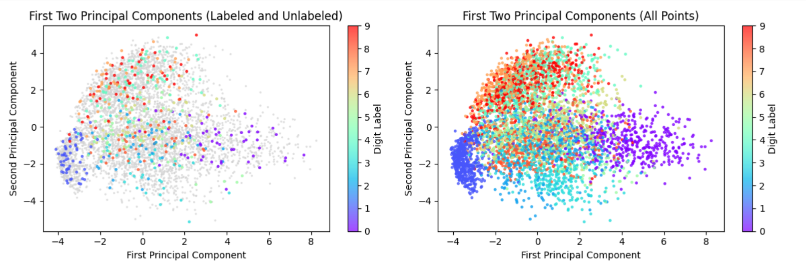
- Time: 4.20 seconds
- Accuracy: 0.976

- Time: 2.89 seconds
- Accuracy: 0.977

MNIST Dataset Preprocessing



- Normalized signal to take value $\in [0, 1]$
- PCA to extract top 50 principle components
- masked 90% of labels to simulate semi-supervised learning; kept full label set for accuracy testing



MNIST Dataset Results

* Ran with A100 GPU runtime on Google Colab
* Parameters:

Using Brute Force kNN:

- Accuracy: 0.933
- Total Time: 18.81 minutes
- Time to construct kNN-graph: 17.14 minutes

Using k Approximate NN:

- Accuracy: 0.955
- Time: 1.79 minutes
- Time to construct kANN-graph: [redacted]

MNIST Dataset Results

* Ran with A100 GPU runtime on Google Colab
* Parameters:

Using Brute Force kNN:

- Accuracy: 0.933
- Total Time: 18.81 minutes
- Time to construct kNN-graph: 17.14 minutes

Using k Approximate NN:

- Accuracy: 0.955
- Time: 1.79 minutes
- Time to construct kANN-graph: **< 6.2 seconds**

MNIST Dataset Results

* Ran with A100 GPU runtime on Google Colab
* Parameters:

Using Brute Force kNN:

- Accuracy: 0.933
- Total Time: 18.81 minutes
- Time to construct kNN-graph: 17.14 minutes

Using k Approximate NN:

- Accuracy: 0.955
- Time: 1.79 minutes
- Time to construct kANN-graph: **< 6.2 seconds**

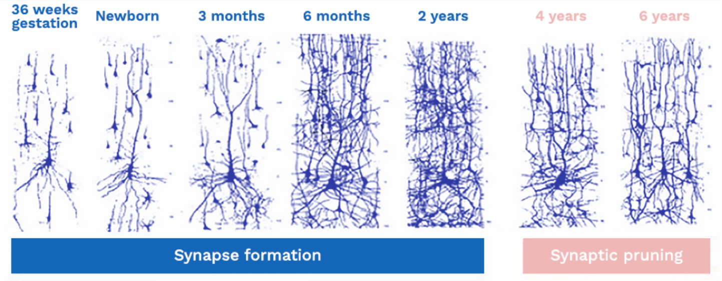
Approximate NN-graph construction is over 165 times faster than the naive

Randomized Pruning in Neural Networks

Why Prune?

- ”It is widely acknowledged that large and sparse models have higher accuracy than small and dense models” [4]
- pruning a model prevents over-fitting
- pruned models also require a fraction of the space a unpruned model would need in memory

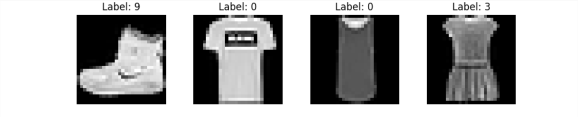
Neurology Inspiration



Synaptic pruning in the brain [2]

Dataset and Neural Network Architecture

- We use a simple architecture recommended by tensorflow
- We prune connections on one layer (more detail on next slide)
- pruning is done by setting pruned weights (outgoing to next layer) to zero
- Inspiration for randomized pruning: "Breaking through Deterministic Barriers: Randomized Pruning Mask Generation and Selection" [4]



MNIST Fashion Dataset

RESULTS

Network	Pruning Strategy	Accuracy
No Pruning	No pruning is applied. All connections are retained.	87.68%
Threshold Fraction Pruning	Prune a specified fraction of connections with the smallest weights.	86.63%
Random Pruning	Prune a random subset of connections, regardless of their weights or importance.	64.56%
Importance-Weighted Random Pruning	Prune connections probabilistically based on their importance, where more important connections are less likely to be pruned.	78.06%
Fractional Random Threshold Pruning	First, select a fraction of connections with the smallest weights. Then, randomly prune a smaller fraction of these selected connections.	87.47%

Summary of Pruning Strategies for Five Neural Networks

Questions???

- W. Dong, M. Charikar, and K. Li.
  Efficient k-nearest neighbor graph construction for generic similarity measures.
  In Proceedings of the 20th international conference on World wide web, pages 577–586. ACM, 2011.
- J. Embrace.
  Synaptic growth, synesthesia, and savant abilities.
  Embrace Autism, 2024.
  Accessed: 2024-08-12.
- S. Har-Peled, P. Indyk, and R. Motwani.
  Approximate nearest neighbor: Towards removing the curse of dimensionality.
  Theory of Computing, 8:321–350, 2012.
  Special issue in honor of Rajeev Motwani.

J. Li, W. Gao, Q. Lei, and D. Xu.
Breaking through deterministic barriers: Randomized pruning mask generation and selection.
arXiv preprint arXiv:2310.13183, 2023.
Accessed: 2024-08-14.

PyNNDescent Developers.
PyNNDescent Documentation, 2024.
Accessed: 2024-08-14.