title: Capstone: Machine Learning From Scratch

author: Daniel Johnston

date: December 7, 2017

Project Description

- **Objective**: of this project is to build a number of known machine learning algorithms from scratch.
- **Parameters**:
    - Restricted to Python, NumPy, SciPy and similar libraries for implementation.
    - Scikit-Learn can only be used to source testing datasets and for performance comparison purposes.
- Metrics:
    - Model Metrics:
        - Regression: R^2
        - Classification: Accuracy
        - Clustering: NA
    - Reach goal: runtimes similar to `sklearn` implementations
- Target Outcomes:
    - Gain a deeper understanding of commonly used machine learning algorithms, with a focus on unsupervised learning and clustering tools
    - Ability to relate algorithms to a broad range of audiences

Model Implementation

Symbol conventions:

- X - the known data used to train a model including both features (columns) and observations (rows)
- X_n - an individual case or observation within X
- x_n - an individual feature within X
- y - the outcomes of known data used to train a model
- y_n - an individual outcome within y

- β_n - coefficient of a fitted model corresponding to the feature x_n
- U - previously unseen data that is run through a model to arrive at a prediction or estimation
- U_n - an individual case or observation within U
- $\widehat{y}$ - the predicted outcomes for U
- $\widehat{y}_n$ - the predicted outcome for the individual case U_n

K Nearest Neighbors for Regression and Classification

Model Description

The K Nearest Neighbors model provides predictions by identifying an arbitrary number, K, of closest neighbors, or observations, in a training set of data to a new observation. Closeness is defined by some measure of distance, such as Euclidean distance.

In Regression applications, the outcomes of the identified K neighbors are aggregated, generally by taking the mean. This aggregated outcome is then applied to the new observation.

In Classification applications, the outcomes of the identified K neighbors are used as votes. The class with the highest number of votes is assigned to the new observation.

Pseudo Code

1. Training data, both features X and outcomes y are stored.
2. For a new observation, U_n, the Euclidean distance between all points of X and U_n are measured.
3. The smallest K distances are used to identify K observations in X.
4. Based on use case:
    1. For Regression: The values of y for the identified K observations are averaged to estimate the outcome, $\widehat{y}$. Example:
        - Where K = 5, for a new case U_n the 5 closest observations have the following values for y_n: 10, 12, 13, 9, 9
        - $\widehat{y}_n = (10 + 12 + 13 + 9 + 9) = 10.6$
    2. For Classification: The values of y for the identified K observations are used as votes to determine the outcome, $\widehat{y}$. Example:
        - Where K = 5, for a new case U_n the 5 closest observations have the following values for y_n: $True, False, True, False, False$
        - Count of $True$ = 2
        - Count of $False$ = 3

        - $3 > 2$ therefor $\hat{y}_n = False$

Testing and Performance

- Regression:
    - Used Boston dataset for comparison
    - R^2 scores
        - From Scratch: 0.639665439953224
        - Sklearn: 0.639665439953224
    - Runtime
        - From Scratch: 2.28 ms ± 79.3 μs per loop
        - Sklearn: 969 μs ± 29.6 μs per loop
- Classification
    - Used Breast Cancer dataset for comparison
    - Accuracy scores:
        - From Scratch: 0.965034965034965
        - Sklearn: 0.965034965034965
    - Runtime
        - From Scratch: 5.83 ms ± 68.2 μs per loop
        - Sklearn: 1.21 ms ± 47 μs per loop

Critique

My implementation relies heavily on the `cdist` function from the `scipy` package. `cdist` takes two matrices and calculates the distance between each point within the two matrices. Sklearn also appears to make use of this function, but appears to utilize sparse matrices which may explain the runtime difference. Additionally, my implementation only has one hyper-parameter, K, while Sklearn has a number of tunable parameters including K, distance metric, and weights. For classification specifically, there is clearly a wider gap in the runtime performance between my implementation and Sklearn's. I suspect that this is due to how the class voting was implemented.

Further development of my implementation would explore the use of sparse matrices and additional distance metrics. For classification uses, I'd like to explore different voting implementations as well as support support for cases in which voting results in a tie.

K-Means Clustering

Model Description

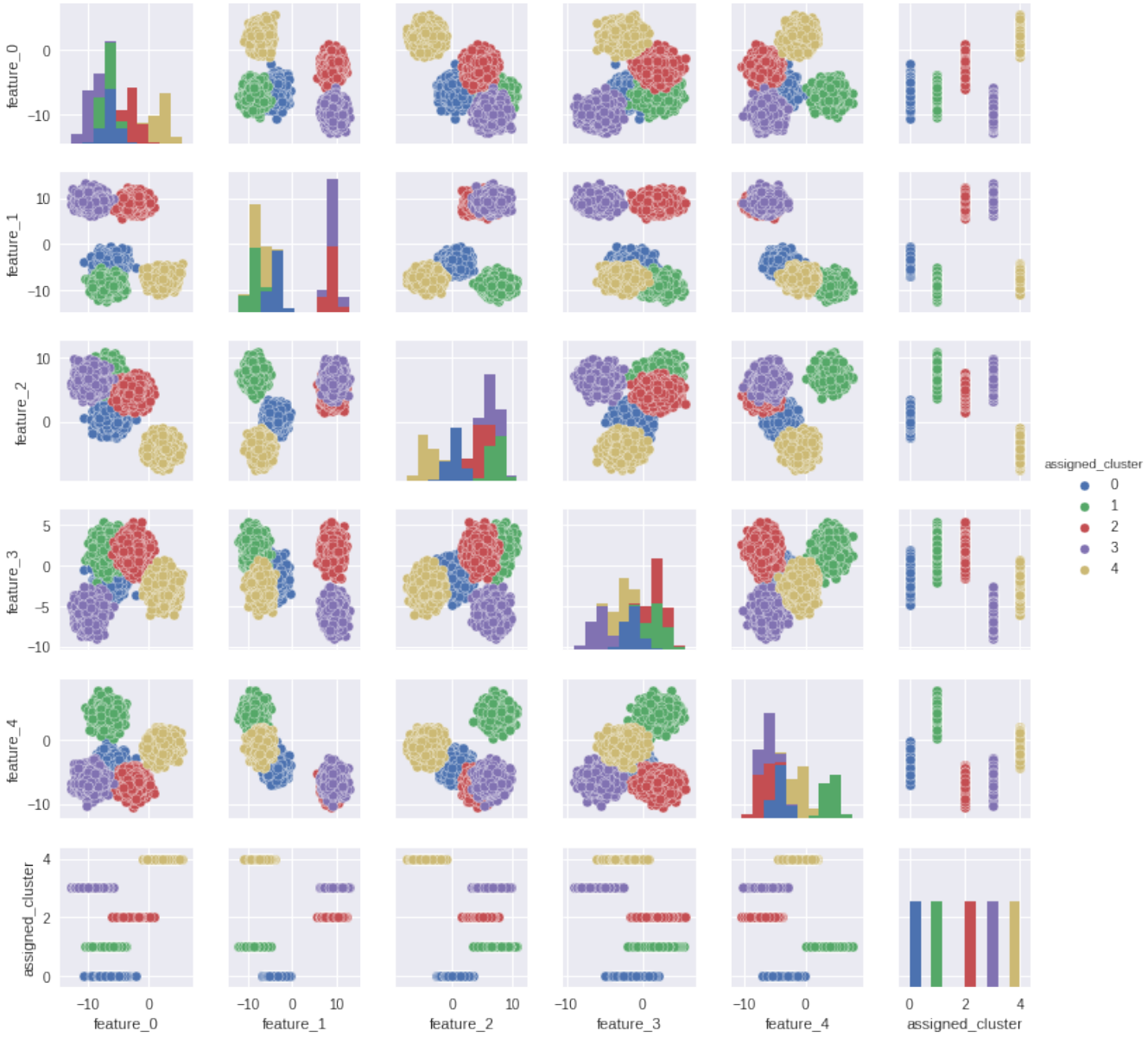
K-Means is an unsupervised machine learning algorithms to identify clusters or segments of similar cases within a set of data. In this case K refers to the number of clusters that the algorithm should find. The algorithm defines clusters by grouping observations into clusters based on the shortest distance to centroids, or the center of each clusters. Once clusters are assigned, the position of each centroid is moved to the centroid of each cluster. Cluster assignment is then reevaluated and centroids are moved again. This is repeated for a defined number of iterations or until some end condition is met. Since the clusters are based on a centroid, KMeans is best used when the data shows natural separation and blob like shapes. Kmeans often fails to be useful in cases where the data has irregular shapes such as half moons.

Pseudo Code

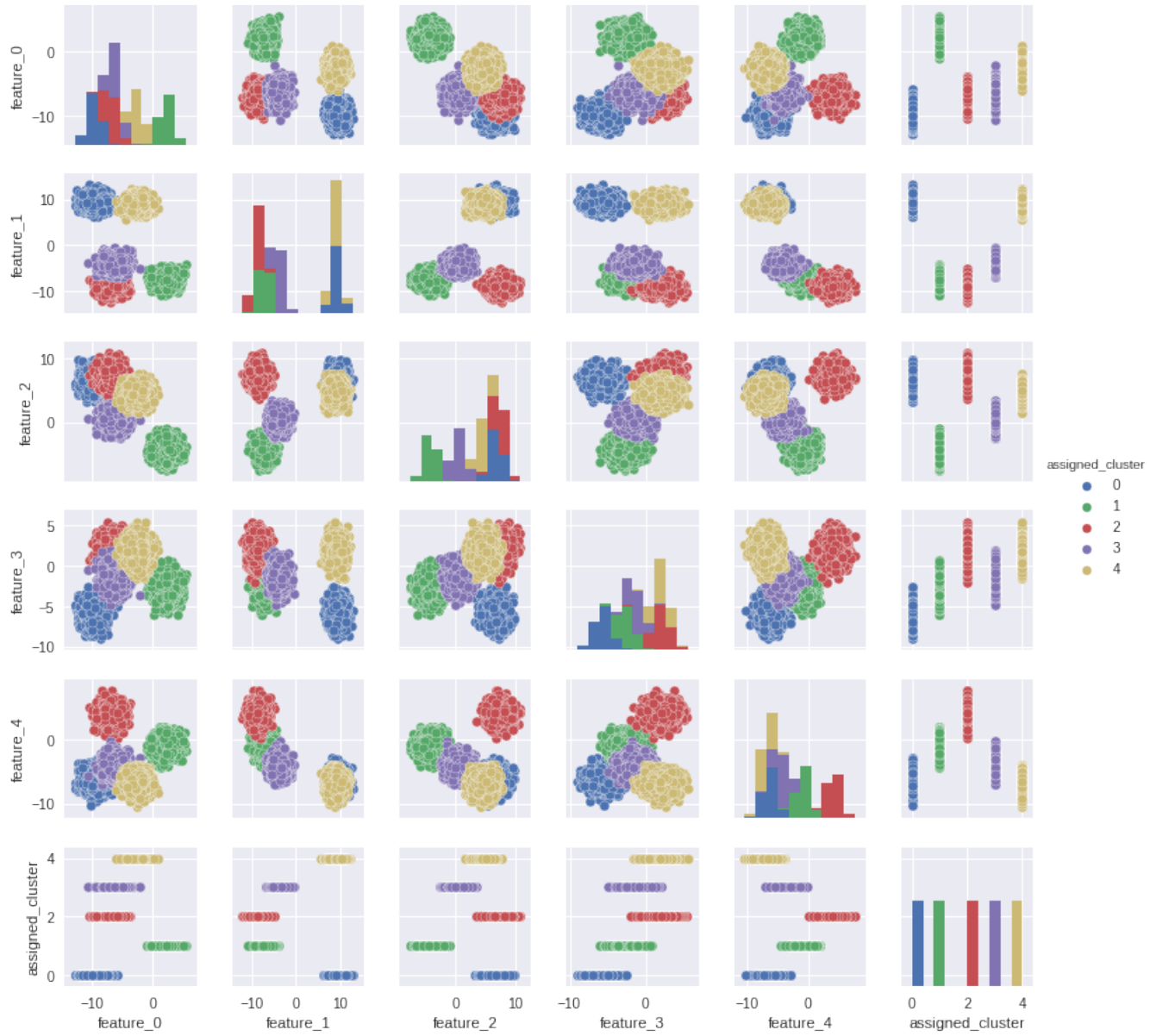
1. Generate K random starting centroids
2. Measure distance between centroids and all observations of X
3. Assign all observations of X to a cluster based on the nearest centroid
4. Move each centroid to the center of the clustered observations of X
5. Repeat steps 2-4 until an end condition is met, such as the cluster assignment remain static, the centroids don't move, or the process has repeated a given number of times.

Testing and Performance

- used `sklearn.datasets.make_blobs` for comparison
    - `make_blobs(n_samples=10000, n_features=12, centers=5, random_state=42)`
- Runtime:
    - From Scratch: 328 ms ± 24.9 ms per loop
    - Sklearn: 64.4 ms ± 1.66 ms per loop
- Cluster: Taking this with a grain of salt since KMeans performance depends so much on the starting points used
    - From Scratch:


feature_0
feature_1
feature_2
feature_3
feature_4
assigned_cluster
assigned_cluster
0
1
2
3
4

- Sklearn:



Critique

What is interesting about KMeans is that the results are largely reliant on the starting position of the centroids. The same set of data can return vastly different results, and suggest different conclusions, so it is often helpful to run a number of different iterations to see how results can vary.

My implementation of K-Mean currently does not have a end condition beyond reaching a user-provided number of iterations. I make use of the `cdist` function to calculate the distance between the centroids and the observations of X. `cdist` works really well for this scenario because it takes in any number of centroids, so it scales well for any giving scenario.

My implementation also only supports random assignment of the starting centroids, based on the uniform random distribution. This can results in starting centroids that will have no observations of X assigned.

Further development would include investigation of different approaches to generating the starting centroids (different random distributions, logical controls, manual definition, etc.)

DB Scan Clustering

Model Description

DB Scan is a clustering algorithm which identifies observations that a clustered together in any irregular shape. It does this by starting with a random observation X_n and then finding other observations within a given radius, r. These observations are then cluster, and the model continues to search outwards to find more observations within r, essential following the path or shape of the data. It continues the search until no new observations are found, at which point it considers the cluster finalized, and starts the process again with a previously unclustered point. Since DBScan identifies clusters based on the the inherent shape of the data, the results should always be consistent across the same set of data, though the clusters may be identified in a different order.

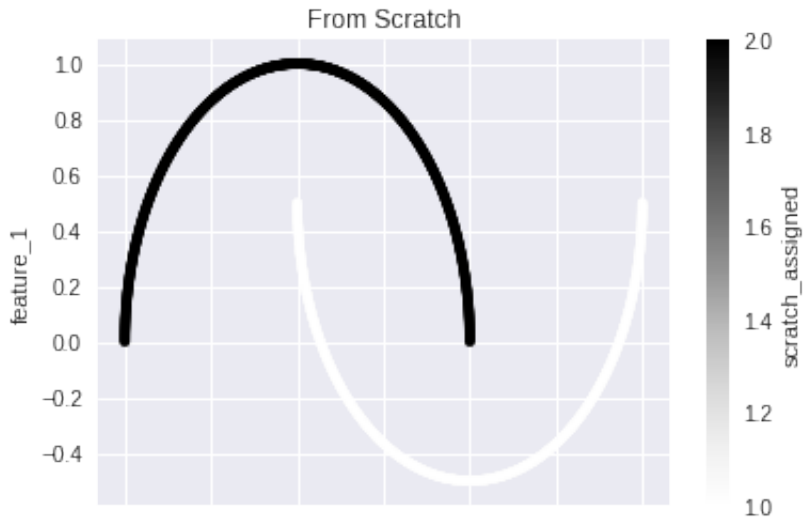
Pseudo Code

1. Define a radius, r as the scanning parameter
2. Pick an unclustered random starting point, X_n , and assign it to cluster c_i
3. Identify unclustered cases of X that are within r distance of X_n . Assign them to cluster c_i
4. Identify unclustered cases of X that are within r distance of any observation within cluster c_i and assign them to cluster c_i.
5. Repeat step 4 until there are no new unclustered cases of X within r of any case within cluster c_i
6. Repeat steps 2-5 until all cases of X are clustered.

Testing and Performance

- used `sklearn.datasets.make_moons` for comparison. This ensured that testing was conducted on irregular data distributions
    - `make_moons(n_samples=1000, random_state=42)`
- Runtime: Sklearn is nearly 10x faster
    - From Scratch: 62.7 ms ± 3.91 ms per loop
    - Sklearn: 7.63 ms ± 310 μs per loop
- Cluster: They are identical
    - From Scratch:

- Sklearn:



Critique

My understanding is that typical implimentations of DB Scan scan for new observations iteratively, i.e. the cluster grows point by point. My implimentation, on the other hand, grows outward in waves, i.e. the outer boundry is expanded by r consitenly until it reaches the bounds of the cluster.

One source of inefficiency with my implimentation is that there are redundant calculations being performed. Further development would work to address this, and possibly change the algorithm to match the point-to-point cluster growth described above. I also use a recursive function to build the clusters which might be less efficient than alternative approaches.

Bottom Up Hierarchical Clustering

Model Description

Bottom Up Hierarchical Clustering starts by assigning every case in X to a separate cluster. Clusters are then joined iteratively, starting with the clusters that are closest together, according to some distance metric, and ending with the clusters that are fartherest apart. The algorithm will eventually end with a single cluster containing all cases of X. The advantage to this approach is realized when you look at the history of how the clusters are joined. This enables the user to identify clusters at any step of the process and choose the ones that are of the most value. A Hierarchal Clustering is that it does not scale well; every case added to X adds a considerable runtime.

Pseudo Code

1. Assign all cases of X to their own cluster
2. Calculate distance between all clusters
3. Group the clusters with the shortest distance
4. Repeat step 3 until all cases have the same cluster

Testing and Performance

- Runtime:
  - `make_blobs(n_samples=10, n_features=2, centers=5, random_state=42)`
    - From Scratch: 93.8 μs ± 2.03 μs per loop
    - Sklearn: 144 μs ± 5.5 μs per loop
  - `make_blobs(n_samples=1000, n_features=2, centers=5, random_state=42)`
    - From Scratch: 13.3 s ± 1.62 s per loop
    - Sklearn: 17.4 ms ± 614 μs per loop

Critique

As seen above, my implimentation does not scale well at all. This is because I create a queue of case-pairs using `itertools` that are then ordered base on the distance measurement. In general working with `numpy` means working without persistant indices. It has been a challenge to work around this limitation but still identify the proper cases to group.

Future development efforts would seek to address this scalability issue. I am unsure of the best solution, but it would likely exclude the use of `itertools`. I'd also like to explore how to convert the cluster history into a dendrogram, which is a visualization to show the itterative clustering.