Project report for kmeanseval

By Ryan Williams

Introduction

K-means is the most common unsupervised learning algorithm used for clustering today. When using k-means, one of the most important decisions to make is how many clusters to use ("k"). This is generally a subjective decision, representing a tradeoff between how simply the clusters summarize the data (small k) vs. how well the clusters represent the data (large k). Picking a k value is typically done by comparing within-cluster-sum of squared errors ("WSS") and/or silhouette scores ("SS") at different values of k. This process often involves creating a chart to visually inspect how WSS and/or SS change at different values of k. This means that every k-means analysis involves some boilerplate code to loop through the creation of multiple k-means models at different values of k, calculate WSS or SS for them, and plot these results.

The goal of the kmeanseval package is to reduce the amount of boilerplate code that goes into selecting a value for k when using k-means. Rather than creating loops to capture WSS and SS and plotting them from scratch, this can all be handled by creating a KMeansEvaluator object with kmeanseval which wraps around the scikit-learn k-means interface to fit models, and then provides methods to get the desired metrics and plots.

Users & Use Cases (Functional Specification)

The intended users for kmeanseval are data scientists, decision scientists, and business analysts who regularly perform clustering analysis with k-means for use cases like summarizing groups of customers or products to be interpreted by humans (implying a low value for k). These kinds of users are likely to frequently employ boilerplate code to find good values for k, which kmeanseval would eliminate – and they also aren't likely to need functionality that kmeanseval doesn't have, like additional evaluation metrics or support for other clustering algorithms. These users are also likely to be familiar with the scikit-learn interface for k-means clustering, which would be valuable (but not required) for using kmeanseval.

kmeanseval isn't designed for users who want to compare multiple clustering algorithms, since kmeanseval only supports k-means. It's also not intended for use cases in which the ideal k may be very large, such as summarizing enormous data sets to ultimately be interpreted by a machine learning model rather than a human; it could still be used for calculating WSS or SS in those cases, but the plotting functionality would become unwieldly (and unnecessary) and the performance may be slow. Finally, kmeanseval isn't designed for users who want to be given an "optimal" value for k. Though some packages offer this - for example by selecting the first value for k that crosses a certain WSS threshold - selecting k is always a subjective decision in my intended use cases and should be done by the users.

The example workflows below both get a list of WSS for k = 2 through 10 and plot the values, demonstrating how using kmeanseval eliminates many lines of boilerplate code even in the simplest scenario.

Workflow using kmeanseval:

```
import kmeanseval
import pandas as pd

kme = kmeanseval.KMeansEvaluator(data = data, k_range = range(2, 11))
wss = kme.get_metrics(method = 'wss')
kme.plot_elbow()
```

Workflow using scikit-learn and matplotlib:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

data = pd.read_csv('data.csv', sep=',')
wss = []
K = range(1, 10)
for k in K:
   model = KMeans(n_clusters=k)
   model.fit(data)
   wss.append(model.inertia_)

plt.figure(figsize=(20,10))
plt.plot(K, wss)
plt.xlabel('k')
plt.ylabel('Within-cluster-sum of squares')
plt.title('WSS for K = 1 through 10')
plt.show()
```

Component Specification

The user-facing functionality for kmeanseval consists of a single module, evaluator.py, which contains the KMeansEvaluator class. Through this class users access all the functionality of kmeansevaluator.

Software Component: *KMeansEvaluator*
A class that wraps a series of scikit-learn KMeans() models. Its public methods offer users the ability to extract and plot evaluation metrics (within-cluster-sum of squared errors, average silhouette score per cluster, and silhouette score per sample) from those models.

User Input:

- *Required:* A dataframe
- *Required:* A range of k over which they want to create models
- *Optional:* Any parameters the user wants to pass to the scikit-learn KMeans models (e.g. defining an initialization method or specifying a random state)

Output:

- A KMeansEvaluator object that contains a scikit-learn KMeans() model for each value in the range of k supplied and has public methods for getting and plotting evaluation metrics, detailed below

Public Methods:

- *get_metrics():* Takes a metric name as an argument (WSS, sample silhouette score, or average silhouette score – defaults to WSS if none supplied) and returns a list of values for that metric
- *plot_elbow():* Takes figure size and text size as arguments, returns an elbow plot of WSS values
- *plot_silhouette_scores():* Takes figure size and text size as arguments, returns a density plot for the distribution of silhouette scores for each sample
- *plot_avg_silhouette_scores():* Takes figure size and text size as arguments, returns a line plot for average silhouette scores for each cluster

Design Decisions

The overarching goal for the design of kmeanseval is to create a simple, lightweight package that can satisfy the majority of use cases for selecting k in a k-means analysis. Most of the design decisions in this project relate to the tradeoff between simplicity and functionality.

- **Algorithm support:** kmeanseval only supports k-means clustering. This does limit the usefulness of the package, however k-means is by far the most popular clustering method. Other clustering methods like GMM, DBSCAN, etc. would add complexity to the code and the interface, requiring support for additional algorithms and evaluation methods, while only serving more niche use cases.
- **Evaluation metric support:** kmeanseval only supports within-cluster-sum of squared errors and silhouette scores as evaluation metrics. Similar to the decision to only support k-means clustering, this simplifies the scope of the project while still providing the needed functionality for the most common use cases.
- **Algorithm source:** kmeanseval wraps around the scikit-learn k-means functions, as opposed to using a different package or doing the clustering from scratch. This will allow any user familiar with using the k-means interface in scikit-learn to easily pick up the interface for kmeanseval.
- **Flexibility considerations:** My goal was to give users the flexibility required for kmeanseval to be useful, and limit flexibility where it isn't needed to simplify the interface and coding requirements. There were a couple of decisions I made specifically related to flexibility:
    - I wanted users to be able to create K-means models by passing any data, and using any parameters, they would be able to use in scikit-learn. To facilitate this I had the KMeansEvaluator class take a **kwargs parameter to pass to the scikit-learn KMeans() function.
    - I wanted plotting functionality to be kept basic, only allowing for changes in plot size and font size. Trying to anticipate all possible plotting needs is too large a task, and users can get the list of WSS/SS values from the KMeansEvaluator object anyway to make their own custom plots if absolutely necessary.
- **Coding paradigm:** kmeanseval is written in an object-oriented coding style. It works by using a "KMeansEvaluator" class which contains data in the form of a series of sklearn KMeans models. The class has methods for returning the values for WSS/SS for those models, or plotting the values.

    - It would have also made sense to use a procedural coding style, given the small scope of this project and the fact that the code rarely needs to be reused across methods (and despite using a class, the way the class works is very "procedural"). A big part of why I chose OOP was to practice coding in a paradigm I'm less familiar with.
- **Separation of concerns:** I identified four user-facing "concerns" that the KMeansEvaluator would need to handle, and gave each one its own public method: one for each of the charts for WSS/Average SS/Sample SS (3 methods total), and one for providing users with a list of any of those metrics. The charts get their own individual methods since each chart works differently, while there's only one function to give users a list of the metric values since the method is performing the same action in every case. Within the plotting functions, I also further separated concerns by moving the "boilerplate" generic starting plot into its own private method. This means each plotting function only needs to call the generic plot and then handle the unique changes it needs to make to that plot. This should make the code more extensible since adding new types of plots and evaluation metrics would only require adding new methods, and not worrying about coupling or interactions.

Comparison: clusteval

Link to clusteval: https://pypi.org/project/clusteval/

clusteval as described on PyPi: *"clusteval is a Python package for unsupervised cluster evaluation. Three evaluation methods are implemented that can be used to evalute clusterings; silhouette, dbindex, and derivative. Four clustering methods can be used: agglomerative, kmeans, dbscan and hdbscan."*

Similar to kmeanseval, clusteval is a tool for calculating and plotting evaluation metrics when clustering data. However, the two packages are functionally very different. clusteval supports k-means clustering, agglomerative clustering, dbscan, and hdbscan while kmeanseval only supports k-means. Both support silhouette score analysis, but only kmeanseval supports WSS, and only clusteval supports davies-bouldin index and derivatives. Both packages wrap around the scikit-learn KMeans() interface for k-means clustering.

Since clusteval supports so many more options, it also has a more complex interface and implementation than kmeanseval. The implementation for clusteval has lots of 'if-then' logic to handle the fact that dbscan has to be instantiated and evaluated very differently from k-means or from agglomerative/hierarchical clustering. This is something I was thinking about when I decided to dedicate my package solely to k-means – if the different clustering methods all require such disparate approaches, it doesn't necessarily make sense to try to handle all of them in one package like clusteval attempts to do.

In terms of usage, clusteval would be more useful for someone trying to compare different clustering approaches – for example, someone trying to decide whether to use k-means or dbscan to cluster their data. For users who have already decided on k-means as an approach, I believe kmeanseval would be more useful than clusteval because of its simplicity and ability to calculate WSS in addition to silhouette scores.

Extensibility

kmeanseval is a simple package and doesn't have any complex coupling or dependencies that would make extending the functionality difficult. The most natural extensions for kmeanseval are additional evaluation metrics and deeper plotting functionality.

Adding support for calculating more evaluation metrics should be a straightforward extension, as only the get_metrics() method would need to be updated with the calculation. The interface would remain simple since this should only expand the options that users can plug into the 'metric' argument. There are no dependencies developers should need to be concerned about when making this kind of update.

However, deeper plotting functionality would make the interface much more complex. Matplotlib allows for iteratively building on plots, but the current format of my code would require all of the options the user wants to be handled by passing parameters to the plotting functions. A function like get_elbow() could be made to take arguments for colors, shapes, line styles, etc., but the real difficulty comes from deciding what's important out of all the options users might theoretically want to plot. A better approach might be creating methods that providing users with a basic pyplot that they can build on with additional matplotlib iterations, as a middle ground between the two approaches currently employed by kmeanseval (providing raw numbers or complete plots).