Design of AI Systems (DAT410)
Assignment 3: AI Tools

Luca Modica
Hugo Manuel Alves Henriques e Silva

February 6, 2024

1 Reading and reflection

The paper *"Hidden Technical Debt in Machine Learning Systems" (by D. Sculley and colleagues from Google)* [3] represents a comprehensive description of the technical debt framework applied to Machine Learning systems. **"Technical debt"** refers to long-term cost associated to quick and easy solution that, in the long-term, can result hard to maintain, costly and even dangerous for the end users. This in ML systems represents a tough challenge: in addition to code issues Sculley argues ML-specific ones at system level, which are more difficult to be detected and harder to solve. The main focus of the paper is to provide a broad perspective of where the technical debt can accumulate faster in ML systems, with the goal of increasing the awareness of the trade-offs that must be considered in practise over the long-term.

- To describe the logic of ML systems the software logic is not sufficient: we also have dependencies on external data. This leads to **weaker abstraction boundaries** (It's more difficult to apply concepts of encapsulation and modular design) and difficulties in making isolated changes. The erosion of those abstract boundaries can increase the technical debt in several ways, from changing the shape of the data (adding or removing features) to leaving a model too widely accessible to the end users.
- ML systems can build technical debt due to **Data dependencies**, which are very hard detect and expensive. In this context, unstable input signals from other data sources and using data that add small to no value to the model can increase technical debt. Copy of a potentially unstable input signal are created, with exhaustive evaluations to regularly remove unnecessary features, can usually help mitigate these issues.
- ML systems are often real-time systems, meaning that their future behavior can influenced by updating them overtime. This leads to **feedback loops**, which are difficult to detect especially if the model is updated infre-

quently. Feedback loops can be generated by future training data (*direct feedback loops*) or even by other systems that communicate (and thus, also influence) each other (*hidden feedback loops*).

- There are also **ML system-design anti-patterns** that, if not avoided or mitigated with code re-factorization, can lead to high maintenance cost and large technical debt. Examples are usage of too many general-purpose packages as black boxes (*glue code*), or too complicated ML pipelines (*Pipeline Jungles*). Possible solutions are to wrap black-boxes libraries code in common APIs, comprehensive data collection and feature extraction.

- Since the external **world is rarely unstable**, we can encounter increasing maintenance cost due to this background rate of change. Setting fixed decision thresholds in the system, or monitoring and testing some of its aspects (prediction biases or action limits), are much more challenging in this case.

Although the usefulness of the technical debt metaphor, the paper precises that it is not a very strict metric to assess the full cost of a Machine Learning system: for that purpose other indicators should be considered, from how easily a new approach can be tested at full scale to how an improvement to the model or data sources can degrade other aspects of the system.

Overall, the Sculley work represents a comprehensive call to action to recognize and address the important challenge of the technical debt. We think this area of maintainable ML is crucial for an ML system to be successful, and for the long-term health of successful ML teams. On the other hand, also the open source community is playing an important role to increase the awareness even more. An example is the the famous Machine Learning library scikit-learn, which also offers comprehensive APIs and documentations to build custom and maintainable estimators [2].

2 Implementation

In this section we will describe a classifier with an underlying KMenas clustering algorithm, implemented by ourselves. This algorithm will be then applied to the popular dataset about PM2.5 Data of 5 Chinese cities ([1]): the goal is to train a KMeans instance to predict the binary variable `PM_HIGH`, which indicates if the pollution level is higher than 100 ug/m$\hat{3}$ (the measurement used is the concentration of the PM2.5 particles). In the end, we will discuss the results obtained by the estimator, eventual encountered issues a possible future improvement.

2.1 Implementation

The proposed solution consists in a Python class that follows the **fit, predict and score pattern**; It's a pattern in the most popular Machine Learning libraries, used to have a common API interfaces among estimators. For this reason, to make sure our implementation will follow these standards, the class inherits 2 Mixin classes from `scikit-learn` package: `BaseEstimator` and `ClusterMixin`.

Hyperparamters and attributes

As every standard estimator, in its initialization the parameter passed represent the hyperparamters of the model: they will then become attributes of the instance. Down below, the available hyperparamters in our implementation are described.

- **n_centers** - Integer that corresponds to the number of desired clusters. Default value is *2*.
- **max_iter** - Integer that corresponds to the maximum number of iterations that we let the algorithm train for. Default value is *100*.
- **init_centroids** - String that corresponds to the initialization method for the centroids. Could be either *"random"* or *"kmeans++"*. Default value is *"random"*.
- **random_seed** - Integer that, if defined, provides deterministic result; the main purpose is to have reproducible environment for testing . Default value is *None.* which means the output will always vary from each creation of the model.
- **distance_metric** - String that corresponds to the desired distance metric. Could be the euclidean distance or the manhattan distance. Default value is *"euclidean_distance.*

The remaining attributes of the class are related to the internal function of the algorithm during training or the output of the algorithm itself. Examples are attributes where the distance function is stored (**distance_func_**), the array of the centroids coordiantes (**centroids_**), and the labels for all the datapoints used for the cluster creation (**labels_**).

Methods

The implemented class contains both private and public methods: some of them are developed for interal logic prupose, others added alos to adhere to the `scikit-learn` standards.

We will now describe the main methods of the KMeans class, in relation to the pattern followed: `.fit()`, `.predict()` and `.score()`.

- **.fit()** - *train the classifier, relying on the KMeans clustering algorithm to identify "k" clusters.* Each of them will have associated a number of datapoints members (the association is stored in the attribute `labels_`) and centroid coordinates.

  There are multiple ways to initialize the centroids, some of which provide better results and convergence than others.

  - As for our implementation, the default centroid initialization selects at random **k** points from the input dataset, each of them corresponding to an initial cluster center.
  - A second initialization option is using **KMeans++** algorithm. After selecting one datapoint for the first centroids the next ones are selected based on a probability distribution, proportional to their squared distance from the nearest existing cluster center. This method is designed to spread out the initial centroids, which often leads to better clustering results and faster convergence in the algorithm.

  After the centroid initialization, we calculate a distance matrix and assign each point to the closest cluster. Each related center will be then updated with the mean of all points that belong to it; if a cluster is empty, a new random datapoint will be chosen as new centroid.

  The above process is repeated until either the maximum number of iterations has been reached or the cluster centers have not changed in consecutive iterations; at that point the labels for each datapoints and the clusters centers coordinates are computed.

- **.predict()** - *predict the clusters (and thus, the labels) of new datapoints.* This by calculating the distance of each instance we want to predict to every centroid, and assign it to the cluster with the closest center of that instance.

- **.score()** - *evaluate the model performance, either in terms of cluster quality or similarity with ground truths labels (if available).* For this purpose, we implemented some characteristic internal and external clustering scoring methods. *Internal methods* evaluate the quality of a clustering structure, without reference to external information; *External methods* instead involve the clustering results to an external set of ground truth labels.

The following are the scoring functions available in our implementation.

- **Inertia** - internal validation scoring method that measures the compactness of the clusters. It calculates the sum of squared distances of each point to its closest centroid.
- **Silhouette Coefficient** - internal validation scoring method that measures how similar an object is to its own cluster compared to other clusters. Values range from -1 to 1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters.
- **Normalized Mutual Information** - external validation method that measures the amount of information obtained about one clustering through knowing the other clustering. In other words, It will measure the *agreement* of the two assignments, ignoring permutations.
- **Adjusted Rand Index** - external function measures the similarity of the 2 assignments, ignoring permutations. Also in this case, the 2 assignments are represented by the labels predicted by the model and a ground truth label set.

It's worth noting that, since the the estimator will be used as classifier, other external scoring function (such as accuracy or f1-score) will be used in the evaluation part.

Sanity check

To assess the quality and the correctness of the implementation, a comprehensive Sanity check and tests were performed.

- The first step is to check the class using the method `check_estimator`, which mainly ensured the estimator adheres to `scikit-learn` conventions. This alongside testing the correctness through an extensive suite of unit tests and python principles in classes.
- Then, the model was tested by generating synthetic data using the method `make_blobs`, always from the `scikit-learn` package. Different data configurations were used, in terms of number of samples, number of centers and the standard deviation of the overall dataset (figure 1). It's worth noticing that the KMeans class was used inside a Pipeline, to also assess its usability in a more production environment.

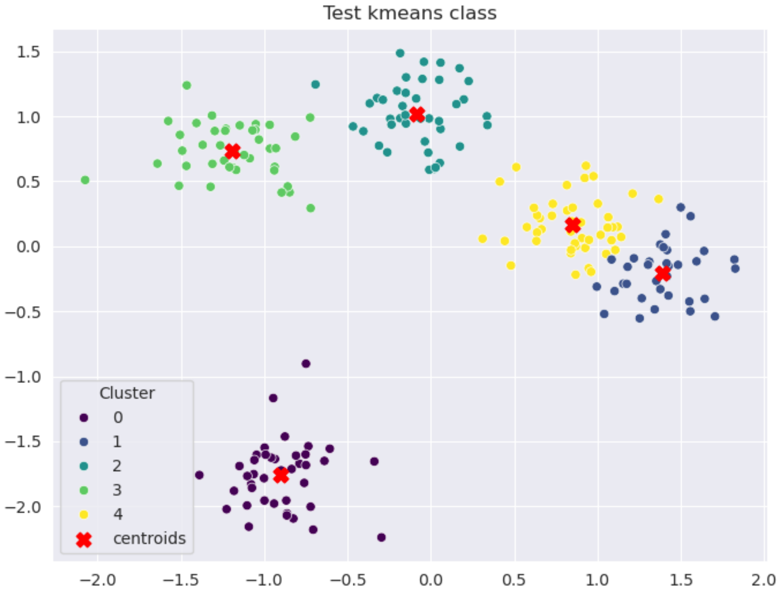

Figure 1: Examples of application of our algorithm on synthetic data.

The visualization of the generated clusters are then followed by the internal and external score checking.

- Lastly, a cluster validation was simulated with data similar to the ones in the previous step. In other words, after removing 20% and 40% of a dataset respectively, It was verified the clusters won't change shape compared to the complete dataset case.

2.2 Application of the estimator

The previously described and tested classifier will be now applied to the Chinese city pollution dataset, to predict the binary target variable **PM_HIGH** from remaining variables.

In particular:

- the system will be trained and validated on the cities **Beijing** and **Shenyang**;
- then, It be individually evaluated on the cities **Guangzhou** and **Shanghai**.

2.3 Training and Evaluation

The starting dataset (with the data of both the cities of Beijing and Shenyang) is composed by 80% of training set and 20% as validation set. Then, since

KMeans is a distance-based model, the datasets were also standardized to take into account its sensibility to different values scales.

After training, the model was first validated by the cluster internal scores, which are the following:

- **Inertia score**: 17196.824183523437;
- **Silhouette score**: 0.26843214299414797.

While the obtained inertia can be considered a baseline for eventual improvements, the silhouette score indicates that the quality of the clusters can be acceptable. This last result was also checked by performing a Principal Component Analysis (PCA) by reducing the dimension of the dataset to 2; in this way, It was possible to plot the generated clusters (even if approximated by the dimensionality reduction) and the related decision boundary (Figure 2)).

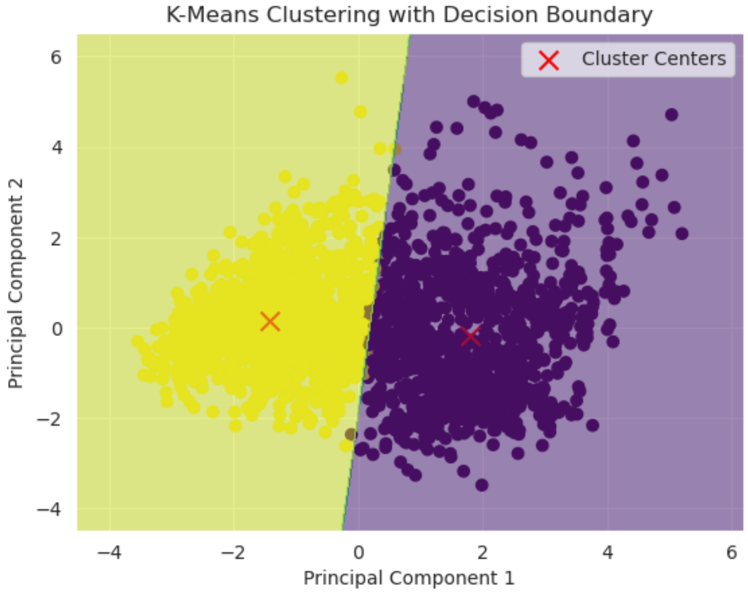

Figure 2: Visualization of the decision boundary with 2 clusters created by KMeans.

As It can be noticed, even though both the 2 clusters are very close to the decision boundary, they are visually separated. This result can represent a good starting point for future improvements, such as experimenting with different clustering models.

The next analyzed metrics are the external cluster scores:

- **Normalized Mutual Info score**: 0.0028264788660817075;

- **Adjusted Rand Index score**: -0.003225155254636967.

The score themselves definitely have room for improvement: both of the scores indicate almost no similarity nor agreement with the ground-truth labels. The reason behind this behavior can be different, depending both from the data and the model we are using: for example, the assumption KMeans makes about the clusters shapes cannot be applied to this specific dataset, which can also be to high-dimensional or can have uninformative features.

Lastly, the tables below show the classification report with the related scores, by using the target variable both for the train set and the validation set:

	precision	recall	f1-score	support
0.0	0.76	0.46	0.57	1687
1.0	0.30	0.61	0.40	629
accuracy			0.50	2316
macro avg	0.53	0.53	0.48	2316
weighted avg	0.63	0.50	0.52	2316

Table 1: Accuracy scores for the train set.

	precision	recall	f1-score	support
0.0	0.74	0.45	0.56	412
1.0	0.30	0.61	0.40	167
accuracy			0.50	579
macro avg	0.53	0.53	0.48	579
weighted avg	0.62	0.50	0.52	579

Table 2: Accuracy scores for the validation set.

Considering the usage of a clustering algorithm for a classification task, also in this case we have acceptable results. This also considering the specific limits of KMeans for more complex dataset, represent a baseline approach to analyze the pollution information.

2.4 Evaluation

Lastly the created model in the previous step was evaluated in 2 different test-sets: the datasets of the cities of Guangzhou and Shanghai. For each of these cities, the classification reports with the most comprehensive metrics are reported below.

	precision	recall	f1-score	support
0.0	0.95	0.10	0.18	1266
1.0	0.07	0.61	0.40	86
accuracy			0.15	1352
macro avg	0.51	0.53	0.48	1352
weighted avg	0.90	0.15	0.18	1352

Table 3: Accuracy scores for the Guangzhou dataset.

	precision	recall	f1-score	support
0.0	0.80	0.27	0.41	1218
1.0	0.05	0.36	0.09	133
accuracy			0.28	1351
macro avg	0.42	0.32	0.25	1351
weighted avg	0.72	0.28	0.38	1351

Table 4: Accuracy scores for the Shanghai dataset.

As in the external cluster scores, the results shown below suffer from the assumption made with the algorithm, alongside the task and the dataset for which is used. As already mentioned, the reason behind scores with large space for improvements can vary a lot: dataset not suitable for KMeasns assumption (spherical-like clusters); high dimensionality noises of the data; ground-truth labels not necessarily correlated to the clusters results, they are not used in the training process.

3 Discussion

- Having better scores in development compared to when actually deploying a system is very common. The main reason is how the model over adapts to the training data and learns its patterns too specifically, so that when it is applied in a new context to new data, it expects to see those same patterns, which could not be exactly present anymore. This is called overfitting. These slight variations might be enough for the algorithm to make wrong predictions and produce a worse score than the one obtained in training. All in all, the algorithm may lack the capability of generalizing to new data, other than the one it has been trained on.

  Our model did show better score in training compared to the evaluation part. It is relevant to remember that we are using a clustering algorithm to classify instances, so it is not entirely "correct" to say that we have overfitting problems. What we can say about our results, is that we are too dependent on the clusters we obtained in training, which can somehow be seen as a way of overfitting. In our implementation we try to optimize the clusters; but since *KMeans* is an unsupervised learning algorithm, we do not try to minimize any error function by comparing to ground-truth labels. We instead "create" new ones, each of them corresponding to a different cluster.

- Assumptions made in our model, which arenrelated to the *KMeans* algorithm are that clusters are isotropic and convex. This essentially means that it works best when the clusters are roughly spherical and have similar variance. If the actual distribution of the high PM2.5 events is not spherical or has different variances, KMeans might not be able to capture the underlying structure well.

- Some of the ways we can improve our solution is, for instance, changing the model to one that takes into account more complex cluster shapes (for example, DBSCAN). Moreover, we can try to reduce the dimensionality of the dataset (by performing PCA), so that some computational burden and possible noise can be removed.

- By the obtained results, it is most likely that the problem above does not meet the assumptions we made, which can be seen in the silhouette score and the accuracy. What is potentially happening is that we are creating spherical clusters to data that doesn't meet a spherical shape.

References

[1] Song Chen. PM2.5 Data of Five Chinese Cities. UCI Machine Learning Repository, 2017. DOI: https://doi.org/10.24432/C52K58.

[2] scikit learn. Scikit-learn custom class standards, 2024. `https://scikit-learn.org/stable/developers/develop.html`, accessed 03/02/2024.

[3] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.