**Introduction:**

In this assignment, I chose to use the same datasets as I did in previous assignments. Both datasets are collections of text snippets. The binary classification dataset is composed of reddit comments – some of them were removed by human moderators while others were not. The multi-class classification dataset is a collection of news headlines and sub-headers classified into one of 20 categories. Both these datasets are interesting because classification today is done manually. By figuring out how to predict classifications for these datasets, we would be able to dramatically reduce the cost of processing large bodies of text. These two datasets are also interesting to compare, since both require vectoring text data, but one is binary while the other is multi-class, resulting in some interesting comparisons between how different models perform.

To start off the upcoming experiments, I used the same feature extraction pipeline I used in prior assignments. I converted each of the text snippets into feature vectors using tf-idf. This allows me to weight commonly encountered words more lightly, while increasing the weight of more uncommon and unique words. Whenever possible, I used sparse matrices to improve performance.

**K-means Clustering**

I applied k-means clustering to the binary dataset using k=2. I chose k=2 because using my existing domain knowledge, I know that I want to predict if the comments were removed or not. Similarly for the multi-class dataset, I chose k=20 since I know that there are 20 news topics being used as classification buckets. Euclidean distance is the default distance formula used for k-means clustering in scikit-learn, so I used that first. I ran both k-means++ and random initialization methods.

Minimizing cluster variation wihin, so weights bigger clusters more. Consider elbow curve

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Homogeneity | Completeness | V-measure | ARI | Silhouette | Accuracy | Time |
| **Binary {n\_clusters: 2}** | | | | | | | |
| Kmean++ | 0.017 | 0.017 | 0.017 | 0.003 | Euclidean: 0.003 Cosine: 0.005 | 0.459 | 56s |
| Random | 0.018 | 0.018 | 0.018 | 0.004 | Euclidean: 0.002 Cosine: 0.004 | 0.458 | 51s |
| **Multi-class {n\_clusters: 20}** | | | | | | | |
| Kmean++ | 0.209 | 0.297 | 0.245 | 0.059 | Euclidean: -0.006 Cosine: -0.017 | 0.048 | 774s |
| Random | 0.237 | 0.317 | 0.271 | 0.068 | Euclidean: -0.006 Cosine: -0.014 | 0.027 | 567s |
| {init: k-means++, n\_init: 10, max\_iter: 300} | | | | | | | |

The results measuring the efficacy of clustering were dismal. My hypothesis was that the data was too dirty for Euclidean distance to handle. Euclidean distance tends to handle outliers poorly – letting any outliers dominate the analysis. To test whether or not that might be true, I printed the top words being used to cluster the text bodies into groups. The words were “the, this, to, is, of, and, it.” Clearly, the model was using the wrong types of words to differentiate between different clusters and was instead choosing the most common words. The Euclidean distance was letting the most common words dominate the analysis. In order to fix that, I enabled the stop\_words function in tf\_idf, where the vectorizer removes the most common words in the English language so that only more unique words remain. Indeed, when pulling the top words again, I saw “people like just study message reddit com www post.” This overall adjustment actually improved accuracy quite a bit, but the clustering metrics still indicate that the clustering algorithm is not working well.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Homogeneity | Completeness | V-measure | ARI | Silhouette | Accuracy | Time |
| **Binary {n\_clusters: 2}** | | | | | | | |
| Kmean++ | 0.000 | 0.000 | 0.000 | 0.003 | Euclidean: 0.001 Cosine: 0.001 | 0.587 | 36s |
| Random | 0.000 | 0.005 | 0.000 | -0.001 | Euclidean: 0.001 Cosine: 0.000 | 0.615 | 31s |
| **Multi-class {n\_clusters: 20}** | | | | | | | |
| Kmean++ | 0.343 | 0.429 | 0.381 | 0.118 | Euclidean: 0.006 Cosine: 0.012 | 0.030 | 496s |
| Random | 0.354 | 0.439 | 0.392 | 0.126 | Euclidean: 0.007 Cosine: 0.013 | 0.077 | 612s |
| {n\_init: 10, max\_iter: 300} | | | | | | | |

One possibility I explored to improve the k-means results was to figure out if the algorithm had fallen to a local minima, since k-means algorithms have a tendency to fall to a local minima. To mitigate this, I tested a couple different values for n\_init, since that parameter introduces randomness in how clusters are initialized. In these tests, I used the random init method since we are trying to introduce more randomness, and that method also resulted in a better accuracy score in the prior experiment. Interestingly, the results were significantly worse as I increased n\_init. This is likely because the algorithm was not at a local minima at all before, and by increasing the number of runs, we actually caused the model to overfit to the data. Although running the model with 30 n\_inits resulted in a slightly better accuracy metric, the increased runtime rendered that iteration less valuable.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N\_init | Homogeneity | Completeness | V-measure | ARI | Silhouette | Accuracy | Time |
| **Binary {n\_clusters: 2}** | | | | | | | |
| 10 | 0.000 | 0.000 | 0.000 | 0.003 | Euclidean: 0.001 Cosine: 0.001 | 0.587 | 36s |
| 20 | 0.000 | 0.001 | 0.000 | 0.003 | Euclidean: 0.001 Cosine: 0.000 | 0.387 | 65s |
| 30 | 0.000 | 0.005 | 0.000 | -0.001 | Euclidean: -0.001 Cosine: 0.002 | 0.615 | 99s |
| **Multi-class {n\_clusters: 20}** | | | | | | | |
| 10 | 0.343 | 0.429 | 0.381 | 0.118 | Euclidean: 0.006 Cosine: 0.012 | 0.030 | 496s |
| 20 | 0.340 | 0.413 | 0.373 | 0.135 | Euclidean: 0.006 Cosine: 0.012 | 0.080 | 1007s |
| 30 | 0.345 | 0.417 | 0.377 | 0.123 | Euclidean: 0.009 Cosine: 0.014 | 0.044 | 2010s |
| {init: random, max\_iter: 300} | | | | | | | |

I reverted n\_init to 10, since increase that parameter did not improve results. I moved onto considering whether or not high dimensionality was impacting results. Particularly with k-means and Euclidean distances, high dimensionality tends to inflate distance calculations. Because text vectorization essentially creates a dictionary of text with a column per word, there are a lot of dimensions that k-means clustering needs to account for. If the data is also sparse, it could further result in outliers that disproportionately impact k-means clustering. To test this, I tried to force-reduce dimensionality by specifying max\_features in the tf-idf vectorizer. The max\_features parameter only selects the top words ordered by term-frequency for consideration. For this analysis, I cared most about the silhouette coefficient, since it measures how well clusters are defined by measuring both intra-cluster and inter-cluster distance. As shown in the table below, reducing the number of features significantly reduced processing time. However, as the number of features grew, we start to observe how increasing dimensionality hurts the overall effectiveness of the clustering algorithm. Interestingly, accuracy actually increases slightly as features increase. My suspicion is that because the dataset leans towards having comments that were not removed, the model is simply choosing the outcome that occurs most often, but has not learned to differentiate between the two binary outcomes. In the next dataset where we run multi-class classification, we can observe if this phenomenon still occurs then. I also pulled the top words which were “people just study don think” for cluster 1 and “like just people feel don” for cluster 2. Notice the overlap in the words being used as features – indeed, the clustering algorithm is not doing a good job of differentiating the binary classes largely because of how the features are modeled.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Features | Silhouette | Accuracy | Time |  | Features | Silhouette | Accuracy | Time |
| 10 | Euclidean: 0.430 Cosine: 0.065 | 0.581 | 0.8s |  | 10 | Euclidean: 0.327 Cosine: 0.406 | 0.046 | 14s |
| 20 | Euclidean: 0.261 Cosine: 0.041 | 0.587 | 1.8s |  | 20 | Euclidean: 0.147 Cosine: 0.212 | 0.049 | 31s |
| 40 | Euclidean: 0.194 Cosine: 0.031 | 0.589 | 2.6s |  | 40 | Euclidean: 0.104 Cosine: 0.194 | 0.052 | 63s |
| {init: random, n\_init: 10, n\_clusters: 2, max\_iter: 300} | | | |  | {init: random, n\_init: 10, n\_clusters: 20, max\_iter: 300} | | | |

I considered using a spherical k-means clustering algorithm, since my suspicion in the binary case is that cosine similarity which measure directional similarity and handles high dimensionality better would perform better for this case. However, since the underlying representation of the features themselves are resulting in significant impact on the model results, I left that analysis for a later point when we handle dimensionality and feature selection.

Check number of components needed here

**Expectation Maximization**

of components As discovered above, the datasets we were looking at overlapped a lot. Expectation maximization actually helps a lot with overlapping datasets. As we saw in the previous section, given simple, well-separated data, k-means finds suitable clustering results.

From an intuitive standpoint, we might expect that the clustering assignment for some points is more certain than others: for example, there appears to be a very slight overlap between the two middle clusters, such that we might not have complete confidence in the cluster assigment of points between them. Unfortunately, the k-means model has no intrinsic measure of probability or uncertainty of cluster assignments (although it may be possible to use a bootstrap approach to estimate this uncertainty). For this, we must think about generalizing the model. These two disadvantages of k-means—its lack of flexibility in cluster shape and lack of probabilistic cluster assignment—mean that for many datasets (especially low-dimensional datasets) it may not perform as well as you might hope.

Under the hood, a Gaussian mixture model is very similar to k-means: it uses an expectation–maximization approach which qualitatively does the following:

The result of this is that each cluster is associated not with a hard-edged sphere, but with a smooth Gaussian model. Just as in the k-means expectation–maximization approach, this algorithm can sometimes miss the globally optimal solution, and thus in practice multiple random initializations are used.

If you look at the details of the preceding fits, you will see that the covariance\_type option was set differently within each. This hyperparameter controls the degrees of freedom in the shape of each cluster; it is essential to set this carefully for any given problem. The default is covariance\_type="diag", which means that the size of the cluster along each dimension can be set independently, with the resulting ellipse constrained to align with the axes. A slightly simpler and faster model is covariance\_type="spherical", which constrains the shape of the cluster such that all dimensions are equal. The resulting clustering will have similar characteristics to that of k-means, though it is not entirely equivalent. A more complicated and computationally expensive model (especially as the number of dimensions grows) is to use covariance\_type="full", which allows each cluster to be modeled as an ellipse with arbitrary orientation.

But if we instead use many more components and ignore the cluster labels, we find a fit that is much closer to the input data:

Expectation maximization can help with overlapping datasets.

Do they otherwise line up naturally? Why or why not? Compare and contrast the different algorithms. What sort of changes might you make to each of those algorithms to improve performance? How much performance was due to the problems you chose?

* Can you describe how the data look in the new spaces you created with the various algorithms? For PCA, what is the distribution of eigenvalues? For ICA, how kurtotic are the distributions? Do the projection axes for ICA seem to capture anything "meaningful"? Assuming you only generate *k* projections (*i.e.*, you do dimensionality reduction), how well is the data reconstructed by the randomized projections? PCA? How much variation did you get when you re-ran your RP several times (I know I don't have to mention that you might want to run RP many times to see what happens, but I hope you forgive me)?
* When you reproduced your clustering experiments on the datasets projected onto the new spaces created by ICA, PCA, and RP, did you get the same clusters as before? Different clusters? Why? Why not?
* When you re-ran your neural network algorithms were there any differences in performance? Speed? Anything at all?

It might be difficult to generate the same kinds of graphs for this part of the assignment as you did before; however, you should come up with some way to describe the kinds of clusters you get. If you can do that visually all the better.

**PCA**

The purpose behind these two algorithms are two-fold. Firstly, the pca algorithm is being used to convert data that might be overly dispersed into a set of linear combinations that can more easily be interpreted. Plot PCA

**ICA**

**Sparse Random Projection**

**Other Feature Selection**

Toggles: n\_components => ‘auto’ using Johnson-Lindenstrauss lemma or int

Might want to graph the results here

np.mean(transformer.components\_ != 0)

^ very few components are 0

Look into downsides and what this does

Approximation technique for distance based method

In mathematics, the Johnson-Lindenstrauss lemma is a result concerning low-distortion embeddings of points from high-dimensional into low-dimensional Euclidean space. The lemma states that a small set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that distances between the points are nearly preserved.

**get\_params**(*deep=True*)[[source]](https://github.com/scikit-learn/scikit-learn/blob/7b136e9/sklearn/base.py#L168)[¶](https://scikit-learn.org/stable/modules/generated/sklearn.random_projection.SparseRandomProjection.html#sklearn.random_projection.SparseRandomProjection.get_params)

states that there exists a mapping from a high-dimensional to a low-dimensional Euclidean space, such that the distance between the points is preserved, within some epsilon variance. Or, simply, the goal is to preserve the pairwise distances between any 2 points in your data.

Maybe also pull covariance?

This is totally unlike methods such as PCA where you typically cross-validate the number of components in conjunction with your classifier.

Consider plotting baseline

· **Reduces Overfitting**: Less redundant data means less opportunity to make decisions based on noise.

· **Improves Accuracy**: Less misleading data means modeling accuracy improves.

· **Reduces Training Time**: fewer data points reduce algorithm complexity and algorithms train faster.

Often, in a high dimensional dataset, there remain some entirely irrelevant, insignificant and unimportant features. It has been seen that the contribution of these types of features is often less towards predictive modeling as compared to the critical features.

* Unnecessary resource allocation for these features.
* These features act as a noise for which the machine learning model can perform terribly poorly.
* The machine model takes more time to get trained.