CS 584: Data Mining Assignment #3. Clustering Iris and Image Datasets

Submitted by William Austin on Aprl 16th, 2020

1. Miner Website Credentials

Registered Miner Username: **waustin3**

2. Final Miner Submissions

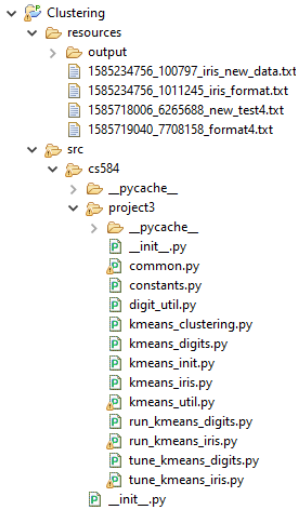
Part	File	V-Score
Part 1 – Iris Clustering	`prediction_waustin_2020-04-14-18-16-06.txt`	**0.86**
Part 2 – Image Clustering	`prediction_waustin_2020-04-15-17-56-57.txt`	**0.70**

3. Instructions for Running the Code

For this project I chose to use Python because of the library support for creaing different models and performing other common machine learning tasks.

My python environment is based on **Conda 4.8.3**, which uses **Python 3.7.4**. The main libraries that I used for the project are:

Library	Version	Website
numpy	1.16.5	https://numpy.org/
scikit-learn	0.21.3	https://scikit-learn.org/stable/



For development, I am using Eclipse with the PyDev module. The project setup is shown to the left.

The project structure is typical for Python applications, with the `src/` directory being the main source directory. All my code is in the `cs584.project3` folder.

The input files with the raw iris and digits data are in the `resources/` directory. My generated output files are in the `resources/output` directory. In my submission, I am including the contents of this directory, but excluding the digits input files, as it is somewhat large.

The main purpose of this assignment was to implement the basic k-means clustering algorithm as well as the bisecting k-means variant and use it to perform clustering on the iris dataset and written digits image dataset. A high-level overview of how my code is structured is below, with the most important files closer to the top:

File	Purpose
`kmeans_clustering.py`	Contains the implementation for k-means clustering and bisecting k-means clustering. Therefore, there are 2 classes: `BasicKMeansClusteringModel`, and `BisectingKMeansClusteringModel`.
`kmeans_util.py`	Utility functions for the k-means implementation. The main 4 functions are for:

	• Calculating the new cluster centers, based on the current cluster assignments. • Calculating new cluster assignments, based on the current cluster assignments • Computing the overall error score (SSE) for each cluster, based on the cluster centers and the current assignments. • Special code to handle the case when an iteration k-means results in an empty cluster (no samples assigned to it).
kmeans_init.py	Generic functions to run on the data set to get the initial cluster centers. I implemented 3 different menthods to do this, as follows: • **initKMeansSampling()** – The standard technique to choose cluster centers from the dataset uniformly, and without replacement. • **initKMeansRandom()** – A variant that simply assigns each point to a random initial cluster and computes the centers of each for k-means initialization • **initKMeansPlusPlus()** – Another better performing variant that tries to choose initial clusters based on the points that are far away from each other.
kmeans_digits.py	Contains functions that perform clustering on the provided digits dataset using various techniques and parameters.
run_kmeans_digits.py	Calls functions in **kmeans_digits.py** to produce an output file for a previously tuned method.
tune_kmeans_digits.py	Calls functions in **kmeans_digits.py** to provide console output for tuning parameters and evaluating the performance of each technique.
kmeans_iris.py	Contains functions that perform clustering on the provided iris dataset using various techniques and parameters.
run_kmeans_iris.py	Calls functions in **kmeans_iris.py** to produce an output file for a previously tuned method.
tune_kmeans_iris.py	Calls functions in **kmeans_iris.py** to provide console output for tuning parameters and evaluating the performance of each technique.
common.py	General functions to perform the following tasks: • Reading datasets into a NumPy array • Writing a generated set of cluster assignments to a formatted output file that can be submitted to Miner. • Generic distance functions that can be used for the k-means algorithm. • Generic center calculation functions that can be used for k-means.
digit_util.py	Image downsampling implementation to convert a handwritten digit record from an array with 784 features to one with 49 features. It does this by transforming the input back into a 28 x 28 matrix and then averages each 4 x 4 patch to generate a 7 x 7 matrix, which is then flattened to an array with dimensions of 1 x 49.
constants.py	Relative paths to input files.

Note that my code for this project is also available on GitHub. The URL is:

https://github.com/william-r-austin/Clustering

4. Approach and Parameter Selection

Each clustering problem for this project can be thought of as being a two-part process. These tasks are (1) feature reduction and preprocessing, and (2) k-means clustering, and each part can be designed and optimized separately. However, both parts need to work well to get good results.

In this section, discuss the approach that I took to accomplish each of these steps for the given data sets. I'll also mentioni main techniques that I experimented with, describing what did and did not work, and if applicable, show data results or where to find the implementation in my code.

1. **Data Representation**
   The first task that I completed for this project was read the input files into NumPy arrays. This is found in the common.py file. I also considered using a basic list to allow for more flexiblity in sample types, as long as the center and distance functions can be provided. However, it makes more sense to preprocess the data into a NumPy array, so this is the format that I used for rest of the project.

2. **Implement the Basic K-Means and Bisecting K-Means Algorithms**
   Both of my k-means implementations try to follow the algorithm from the class slides as closely as possible. The basic k-means algorithm is below.

```
1: Select K points as the initial centroids.
2: repeat
3:     Form K clusters by assigning all points to the closest centroid.
4:     Recompute the centroid of each cluster.
5: until The centroids don't change
```

   In my code, this functionality is implemented by the **`BasicKMeansClusteringModel`** class in the **`kmeans_clustering.py`** file. A callback function reference is passed in for the the initialization step. The most common of these methods are implemented in **`kmeans_init.py`**. In addtion, callers can use a lambda to specify a different method or customize the behavior of one of the existing methods (by using a different distance metric, for example).

   Each of the two main steps inside the loop is implemented by a helper function in **`kmeans_util.py`**. This makes the high-level code that is part of the class implementation very simple. The only additional step not shown in the pseudocode is to handle empty clusters that may occure. We simply reassign one of the samples from the worst cluster (according to the SSE score) to the empty cluster.

   Next, we will look at the bisecting k-means algorithm:

```
1: Initialize the list of clusters to contain the cluster containing all points.
2: repeat
3:     Select a cluster from the list of clusters
4:     for i = 1 to number_of_iterations do
5:         Bisect the selected cluster using basic K-means
6:     end for
7:     Add the two clusters from the bisection with the lowest SSE to the list of clusters.
8: until Until the list of clusters contains K clusters
```

   Like the basic k-means, my bisecting k-means is implemented in the **`kmeans_clustering.py`** file. The class is **`BisectingKMeansClusteringModel`**. As specified in the assignment, this class calls my implementation for the basic k-means step.

   To call either one of these classes, the constructors take the following parameters to initialize the class:

   - **`distanceFunction`** – Function reference that takes 2 sample and returns the distance between them. The arguments will be samples from the dataset to be clustered (1-D Numpy arrays in this case).
   - **`centerFunction`** – Function reference that takes an array of samples and returns the center. We use the average of all the samples, but could provide a variant that used the median, for example.

- **`maxIterations`** – For the basic algorithm, this is an upper limit on the number of iterations because the convergence may be very slow. For the bisecting algorithm, this is the number of times to run the basic k-means algorithm as part of the bisecting operation.
- **`initFunction`** (basic k-means only) – Function reference to create a list of initial cluster centers. It's not needed for the bisecting version because all samples are initially added to the same cluster.

After initializing the model object with these parameters, we can pass the dataset and cluster labels (represented by NumPy arrays) to **`runBasicKMeansClustering()`** or **`runBisectingKMeansClustering()`** to execute the algorithm.

Lastly, in both cases, after the operation, we make a number of variables available:

- **`finalClusterCenters`** – A dictionary with the cluster label as the key and the cluster center (a NumPy array) as the value.
- **`finalClusterAssignments`** – A NumPy array of cluster labels matching the size of the given dataset.
- **`finalClusterErrorMap`** – A dictionary mapping each cluster label to the cluster quality. As expected, I used the SSE (sum of squared errors) for this metric, which sums the squared distance for each sample in the cluster to the cluster center. See the next section for details.
- **`finalClusterErrorTotal`** – The sum of all individual cluster errors, which we can use to estimate how well the clustering worked.

3. Implement Metric for Evaluating Cluster Quality

As expected, I used the sum of squared errors metric for this metric. The formula given in the slides is:

$$SSE = \sum_{i=1}^{K} \sum_{x \in C_i} d(m_i, x)^2$$

where there are K total clusters, m_i is the center of cluster C_i, and x is a sample assigned to cluster C_i.

In my code, use dictionary to store the SSE for each individual cluster, and the global SSE value is simply the sum of these values. The code is:

```
def computeClusterErrorMap(X, clusterAssignments, clusterCenters, distanceFunction):
    clusterErrorMap = {}
    for clusterLabel, clusterCenter in clusterCenters.items():
        func = np.vectorize(lambda t: t == clusterLabel)
        indices = func(clusterAssignments)
        clusterPoints = X[indices, :]
        squaredErrorFunc = lambda t: distanceFunction(t, clusterCenter) ** 2
        sseArray = np.apply_along_axis(squaredErrorFunc, axis=1, arr=clusterPoints)
        totalClusterSSE = np.sum(a=sseArray, axis=0, dtype=np.float64)
        clusterErrorMap[clusterLabel] = totalClusterSSE

    return clusterErrorMap
```

We are also asked to look at the impact of cluster size on this metric. When I ran my experiment with the image data for cluster size from 2 to 20 in steps of 2, I got the following output:

Unfortunately, this graph does not show the typical "corner" at K = 10, but it does show an overall decreasing trend.

4. Choose Dimensionality Reduction and Preprocessing Techniques

Because each dataset represents vastly different information, the dimensionality reduction and preprocessing needs to be different for both, to achieve the best results. Therefore, I will discuss them datasets separately:

Part 1. Iris Dataset

For the Iris dataset, there are only 4 features in the dataset, so we don't need to reduce it further. However, there is some preprocessing that we need to do to normalize the dataset. To achieve this, we use the sklearn.preprocessing.StandardScaler, which translates the data so that the mean is 0 and scales it so that the variance is 1. Just doing these simple steps was enough to produce good results on the Iris dataset (0.86 v-measure score on Miner).

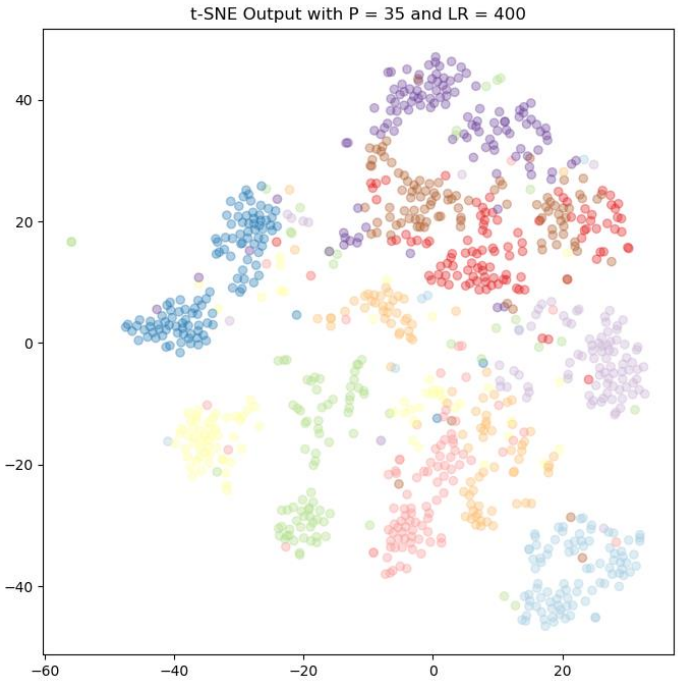
I also tried to use t-SNE on the Iris dataset using close to the recommended default parameters, but the performance was not good, and the Miner v-measure score dropped to 0.65.

Part 2. Written Images Dataset

Obviously, the second dataset in this assignment is much more complicated than the first. Therefore, I tried a few different approachs. My first idea was to do feature reduction by downsampling. This is implemented in the **`digits_util.py`** file and it transform the 1x784 feature vector for each sample into a 1x49 feature vector by interpreting the data as a 28x28 image and downsampling to a 7x7 image by averaging each 4x4 patch and flattening it back to a row vector. My first submission simply used this reduced dataset with the basic version of k-means, resulting in a Miner score of 0.40.

Next, at the suggestion of Dr. Lin, I investigated the t-SNE (t-distributed Stochastic Neighbor Embedding) functionality, available through the **`sklearn.manifold.TSNE`** class. In order to improve the documentation recommends reducing the number of features to about 50 before executing the t-SNE procedure. However, when I combined t-SNE with PCA and basic k-means clustering, my Miner score dropped to 0.38 for the second submission.

Because I needed to better understand how the t-SNE algorithm was working, I downloaded a labeled MNIST dataset. Note that the only thing that I used this for was for creating charts to better understand the parameters and behavior of t-SNE. It was not used to bypass the omission of labels in the given dataset. The main parameters that I played with were the **`learning_rate`** and **`perplexity`**. I found that a learning_rate of 400 and a perplexity of 35 (approximately $\sqrt{N}$, where N is the input data set size) seemed to work well. Here is a sample chart that I created with the colors representing the digit label:



Using these parameters with t-SNE on the raw digit feature vectors seemed to work better, and eventually led to my final Miner submission v-measure score of 0.70. Note that I also continued trying to experiment with combining t-SNE with PCA and downsampling, but my results did not improve.

5. Results and Observations

Over the course of this project there are many different techniques and parameters that I investigated. In this section, I will give an overview of how these factors impacted the results.

- **K-Means Type:** For both datasets, I executed the clustering with the basic k-means model as well as the bisecting k-means model. For the Iris dataset, I performed 5 trials for each with random sampling for the initialization, and normalized inputs. In both cases, the best SSE score was the same (0.04618), which gave me a v-measure score of 0.86 on Miner. However, I did notice 2 minor differences:
  - When using t-SNE with the Iris dataset, the best clustering produced by both methods had the same SSE, but *average* SSE produced by the 5 trials of bisecting k-means was better. (2656.3 vs 3482.9 for basic)
  - The basic implementation runs somewhat faster. However, this is highly dependent on how many iterations of the basic algorithm run inside the loop for the bisecting algorithm. In the slides, this is the number_of_iterations variable.

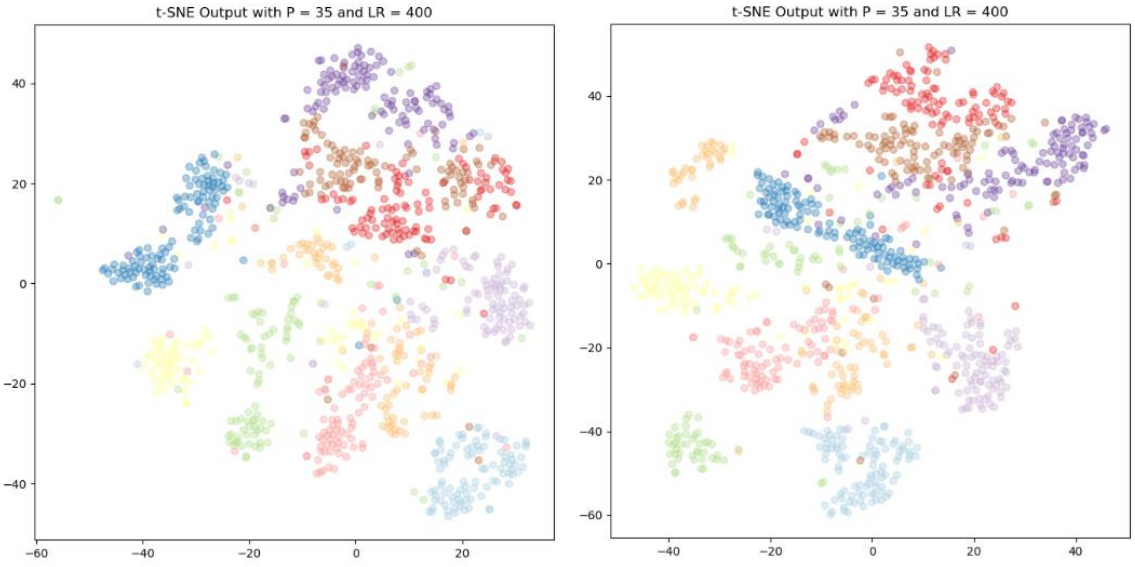
  For the Image dataset, I also tested both k-means types with my t-SNE implementation using a best-of-5 trials setup. For the basic k-means version, the best SSE was 4658495.448, and for the bisecting version, the best was 5605174.124. Therefore, the basic version was better by a significant amount, so I used the results from that execution to submit to Miner for a v-measure score of 0.70. However, I noted that the running time was about the same in both cases because I used fewer iterations for the bisecting version.

- **K-Means Initialization Type:** I compared the impact of the initialization type on the results using the Iris dataset. The results are shown in the table below. For each type, we report on the best SSE result and the average SSE result over 5 trials.

Method	Best SSE Result	Average SSE Result
Sampling: Uniformly choose K centers from the sample points.	140.97	152.81
Random: Uniformly assign each sample to a cluster and report the resulting centers	142.11	142.11
K-Means++: Use the K-Means++ to try to choose "good" centers that are not close to each other.	140.97	141.51

As we can see, the sampling approach can produce a good result (matching the best reported in this case), but there is a wide variation, and many results are very bad. Random sampling got stuck in the same local minimum for all 5 trials and did not find the best result. K-Means++, on the other hand consistently produces good results and found the overall best SSE result as well.

- **Image Preprocessing by Downsampling:** I did not do extensive testing with downsampling after my initial submission. However, it did not seem to change the quality of the clustering a great deal. The two images show the comparison of running t-SNE on the raw data and the downsampled data.

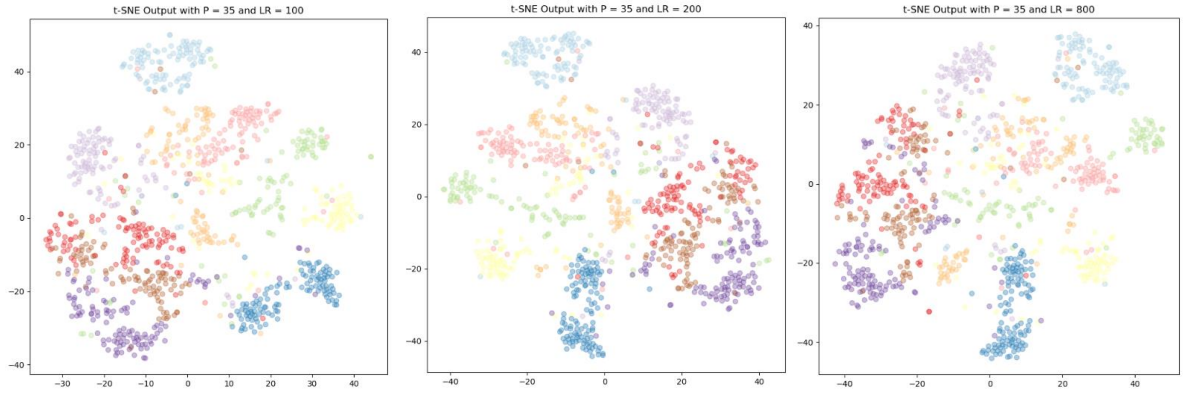


No Downsampling (1x784 input) After Downsampling (1x49 input)

- **t-SNE Perplexity:** For t-SNE, the perplexity parameter relates to the number of neighbor points that influence the transformation. For my experimenting, I looked at the clustering results for several different perplexity values. These are shown below. In the end, I ended up using a perplexity value of 35. However, some sources recommend setting it to approxmately $\sqrt{N}$.

Perplexity = 5 Perplexity = 45 Perplexity = 100

- **t-SNE Learning Rate:** Again, as recommended by the TSNE documentation, I experimented with learning rates between 100 and 1000 and ended up choosing a value of 400 for my submission. The charts below show the impact that the learning rate has on the clustering quality.



Learning Rate = 100 Learning Rate = 200 Learning Rate = 800

- **t-SNE Components:** The last t-SNE parameter that I experimented with was the number of components. Setting `n_components` = 2 is by far the most popular option because the results are easy to plot and visualize. However, there is "more room" in a three-dimensional space to separate and cluster points, so I briefly tried setting `n_components` = 3. Unfortunately, my results were inferior to what I had been producing before. I believe that the reason for this was because there is an impact on the best values for the other parameters if the number of components is changed, so for my submission file, I left the value as 2, which is also the default.

6. Algorithm Complexity and Runtime

As noted in the slides, the basic k-means algorithm is $O(tknd)$, where t = # of iterations, k = # of clusters, n = #of samples, and d = # of features. Obviously, the number of iterations can vary across trials, based on the initial clusters, but in practice, I found that the algorithm converged in less than 10 steps for the Iris dataset, and less than 100 steps for the Images dataset. In addition, the runtime for bisecting k-means will be similar, because we are running basic k-means K times, but only on about a sample size of $\frac{i}{K}$ for cluster i, so over the whole algorithm, which works out to about $\ln(K)$, so the overall running time of bisecting k-means will be $O(tnd \ln k)$.

Note that for the purposes of this project, the time for running t-SNE far exceeded the time for k-means. For a typical execution on my machine (Dell Latitude E5470 laptop with 16 GB ram and a dual core 2.3 Ghz CPU), the t-SNE step usually took about 15 minutes for the Images dataset, and an execution of k-means would take 1-2 minutes.

7. Lessons Learned and Other Techniques

Briefly, I'll outline a few of the takeaways from this assignment:

- Empty clusters are a real problem for k-means. Before I added code to handle this case, this caused issues in my code because I was trying to compute the distance to a center that did not exist. Therefore, I added code to fix the situation using one of the more popular methods (take a sample from the worst cluster and assign it to the empty cluster). However, it would also be possible to define another parameter to allow the user to define a different way to handle the empty clusters
- Although we can investigate different clustering approaches using the SSE metric for a single data set, it is often not possible to compare these results directly when using a different preprocessing technique. For example, PCA and t-SNE will transform the data set in different ways, so the resulting comparison will be invalid.
- One of the challenges with this dataset is that we were tasked with implementing a clustering algorithm, which is an unsupervised learning approach. In previous projects, we could use cross-validation on the training set to try to improve the model. However, in this case, our clustering metric (sum of squared errors) may not give a useful indication of how good our preprocessing steps are. Additionally, even if our cluster density is improving, without labels, we may not know if we are identifying the *right* clusters, and there would be no corresponding increase in the v-measure score.