INFO I-535 Course Project:
Distributed K-Means and Image Clustering

William McWhorter

Contents

1 Introduction

Clustering is an unsupervised machine learning technique where data instances are grouped together based upon common characteristics in the feature space of these instances. This method of unsupervised learning has many applications, such as exploratory data analysis, customer segmentation, visual searches, and fraud detection [2]. Clustering algorithms can be a helpful first step when analyzing a dataset for the first time, even if the data is labeled, just as a form of data exploration.

There are many different types of clustering algorithms, for example, hierarchical clustering (such as agglomerative clustering), density based clustering (such as DBSCAN), and squared-error based clustering (such as K-Means) [5]. In the case of K-Means clustering, K groups are selected, points are assigned to these groups based upon their proximity to the center point, or centroid, of the group. The centroid is updated as the mean location of all members of that group, and the process is repeated until convergence of the centroids has been achieved.

2 Background

Clustering can be seen as a helpful, if not important, first step when analyzing new data. One common method of clustering which might be applied, is K-Means clustering. One reason for this could be its simplicity and the ability to easily understand the algorithm. However, like anything, comes with disadvantages such as non-interpretability of the clusters, high sensitivity to initialization and noise, and inability to deal with clusters that vary in size or density [3]. Because of K-Means being such a popular method for clustering, and with the rate of increase in the size of data, it is important to be able to operate the algorithm in a distributed framework. As such, this project dealt with implementing a generic K-Means algorithm, both with random initializations and with K-Means++ initialization (new centroids are chosen probabilistically, with points far from the already chosen centroids being more likely to be selected), in a distributed framework, so that when data scales too large to fit into a single machine it can still be operated on. Once the algorithm was implemented, it was tested on a set of approximately 2600 images of weather conditions on Kaggle [4].

3 Methodology

To begin addressing the problem of implementing a distributed K-Means, a virtual machine comprised of two CPU cores, six gigabytes of RAM, and 30 gigabytes of disk storage running on Ubuntu 22.04 was created using the Exosphere virtual machines provided by Jetstream2. As far as software is concerned, the K-Means class was implemented entirely in Python, primarily utilizing base Python, as well as some basic functions from Numpy, as well as PySpark to handle the nuances of distributed programming such as job distribution and collection and fault handling. To be more specific, the class was implemented using Pyhton version 3.11.6, Numpy version 1.24.4, and PySpark version 3.5.0. For the Spark Context, the configuration was set to have the driver to have 3 gigabytes of memory, the executors to each have 1 gigabyte of memory, and the driver to have a max result size of 2 gigabytes. All other configuration attributes were left at their default values.

The structure that was chosen was to implement a K-Means class in python roughly mimicking the Lloyd's algorithm version of the K-Means class implemented in the Sci-Kit Learn's clustering library. This involved constructing methods that would allow a user to run multiple random initializations using the standard Lloyd's algorithm and automatically returning the best set of centroids, as well as implementing the K-Means++ initialization methods. A predict function is also built into the class so that a trained model can report the clusters to which new data would belong. There are also attributes for the total cluster inertia so that a user can attempt to determine an optimal number of clusters via the elbow method of determination, as well as attributes to report the number of clusters, the cluster centroids of the last fit, the number of iterations needed to converge, and the number of features detected in the training data. The code for the implementation of the class can be found in Appendix A.

In order to test the K-Means class implementation, it was tested on both the popular iris dataset for a small and simple example, as well as a larger scale set of images so that the data it was run on would not simply fit into the memory of the machine. The iris data simply needed to be converted to a PySpark RDD using `PySpark.SparkContext.Parallelize`, and then passed in with a zipped index as described in the class documentation. For the weather images dataset, however, the images first needed to be preprocessed by converting them all to RGB images rather than RGBa, and any images that were greyscale were removed. The images were also converted to be `uint8` format rather than `float32` format to minimized and memory and storage requirements throughout the process. For similar reasons, the images were scaled to the mean image size. Finally the images were flattened so they could be processed by the clustering algorithm. For simplicity, the images were also reduced to only 5 of the original image classes. Specifically, only images under the classes `Lightning`, `Rain`, `Rainbow`, `Sandstorm`, and `Fogsmog` were kept. All of the image preprocessing steps were done via PySpark, as well, when possible. The images were kept in two RDDs. One RDD stored the image names as values with an index ranging from zero to the number of images minus one, and the other stored the image data as flattened arrays as the values with the same index as mentioned prior, such that the RDDs could be recombined later via the index. The code for the image preprocessing can be found in Appendix B.

The full code for the implementation, testing, and application can be found posted to GitHub [1].

4 Results

In an attempt to ensure that the distributed K-Means class is working properly, it was preformed on the famous iris flower dataset for a simple and easy to process example. The results of the iris data on the distributed K-Means class were compared to the results of Sci-Kit Learn's K-Means class to determine if the class was implemented properly. The confusion matrices shown in Table (1) and Table (2) were obtained by taking the correct labels and then all permutations of the predicted labels to maximize the clustering accuracy.

	Predicted		
Actual	Setosa	Versicolor	Virginica
Setosa	50	0	0
Versicolor	0	47	3
Virginica	0	14	36

Table 1: Confusion matrix distributed K-Means clustering.

	Predicted		
Actual	Setosa	Versicolor	Virginica
Setosa	50	0	0
Versicolor	0	48	2
Virginica	0	14	36

Table 2: Confusion matrix for Sci-Kit Learn K-Means clustering.

Based upon these results, it can be seen that the two methods return almost identical confusion matrices, and hence clustering accuracies (88.67% and 89.33%, respectively), and likely only differ because of randomness in the selection of initial centroid locations, even with the `K-Means++` initialization method. As such, it should then be safe to move on to testing the distributed K-Means class on a dataset that is too large to fit into the machine's memory concurrently.

By running the distributed K-Means class on the images dataset for a varying number of clusters (specifically, 1 through 10 clusters), and recording the total cluster inertia, an inertia plot was able to be produced and used to try and determine and optimal number of clusters. This can be seen in Figure (1).

Using the elbow method of determination, an optimal number of clusters can be set seemingly anywhere between roughly 3 and 5 clusters. However, the true optimal number of clusters should be 5, as this was the number of classes that the datasets was trimmed to in the preprocessing stage. Due to this fact, a set

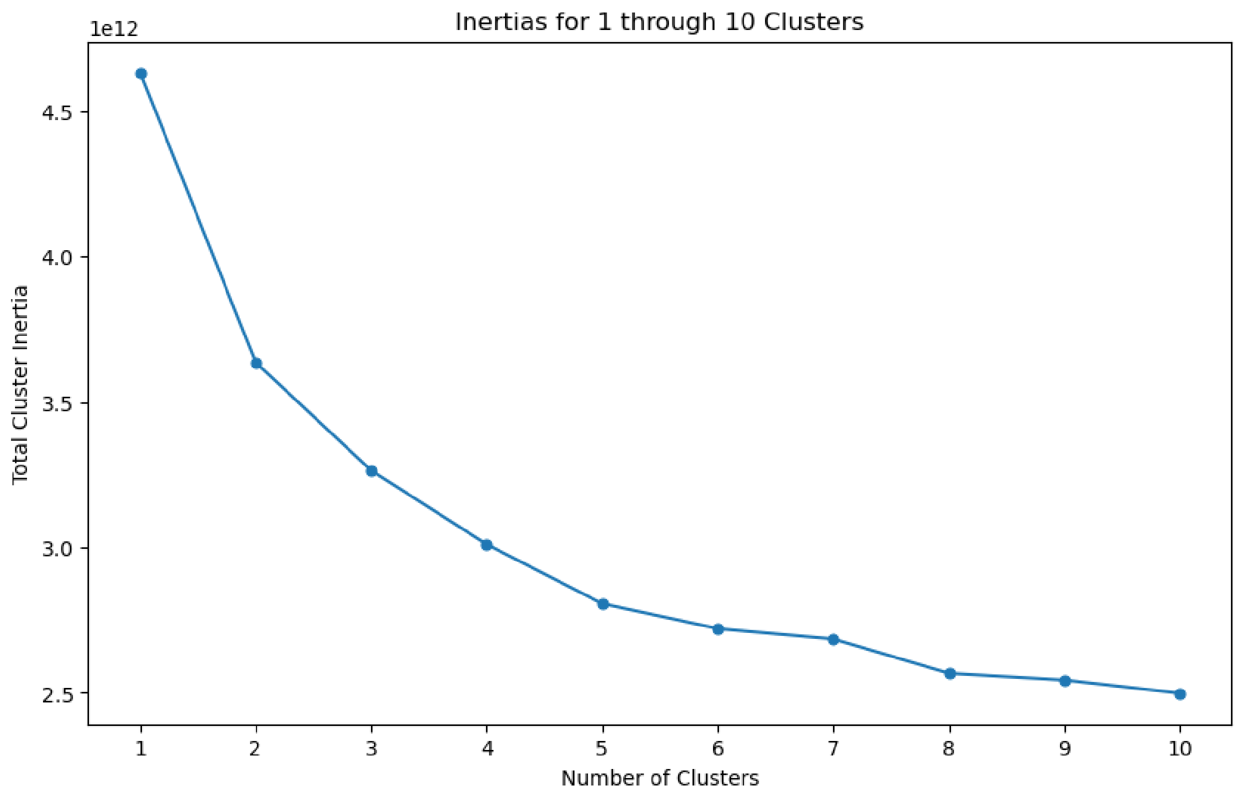

Figure 1: Plot of the total cluster inertia across a range of different cluster counts using the weather images dataset.

of centroids was generating with setting the number of clusters equal to 5. The confusion matrix of the resulting predictions is shown in Table (3) and a set of sample images paired with a visualization of each classes' assigned cluster centers are shown in Figure (2). The clustering accuracy can then be found to be approximately 50.27%.

	Predicted				
Actual	Rain	Fogsmog	Sandstorm	Rainbow	Lightning
Rain	319	75	3	48	75
Fogsmog	57	384	7	339	28
Sandstorm	36	213	274	121	41
Rainbow	67	55	1	89	20
Lightning	64	43	0	13	254

Table 3: Confusion matrix for the clustering of the weather images dataset.

5 Discussion

5.1 Interpretation of Results

Based upon the confusion matrices for the iris data, it seems that the distributed K-Means class was implemented properly in terms of results. The ability to operate across multiple processors using PySpark demonstrates that the class was successfully implemented in a distributed manner. Thus, the goal of implementing a distributed K-Means clustering class, capable of operating agnostic of the data provided to it, assuming the data is input in the appropriate manner, appears to have been achieved.

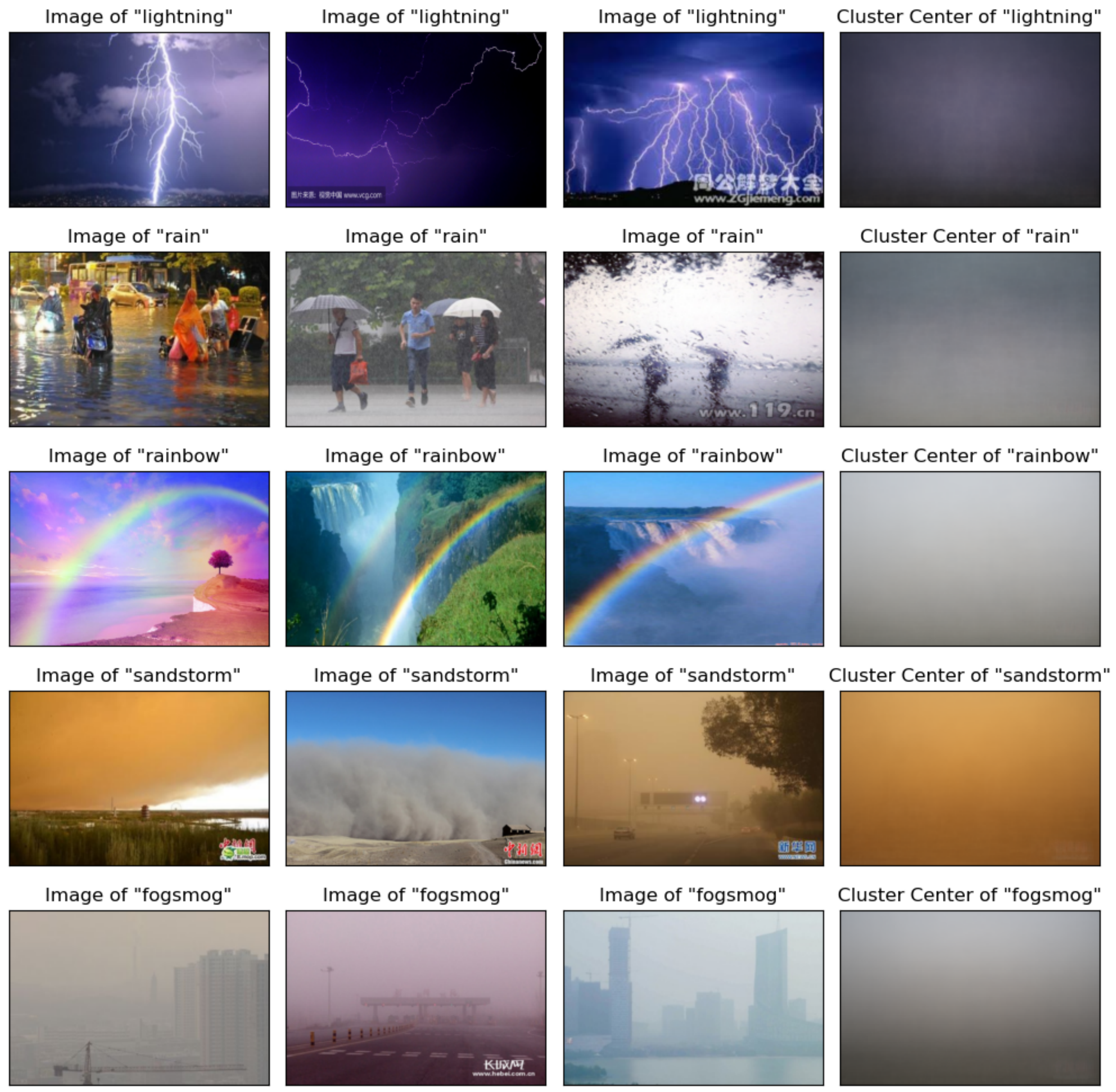

Figure 2: Visualizations of sample images and the visualization of the cluster center assigned to the class of those images.

As far as the performance of the distributed K-Means clustering on the weather images dataset is concerned, it performed better than expected. K-Means is not inherently well suited for images, as K-Means operates by updating centroid locations via the average of all instances within a cluster at each iteration, where in the case of images, each feature of the instances is a specific color channel of a specific pixel within the images. Because of this, the resultant centroid locations are simply the average color of each pixel throughout the image, which can be seen in Figure (2). The determination of the number of optimal clusters, after seeing the visualized centroids, tracks as it can easily be seen how the algorithm might want to group instances of `Rain`, `Rainbow`, and `Fogsmog` into one or two clusters, rather than three separate clusters, thus resulting in the other values of the estimated 3 to 5 clusters. The off-diagonal values in the confusion matrix of Table (3) also tend to support this evaluation.

This results could potentially be improved upon, however. One possible method for improving the results is to reduce the dimensionality of the images so that K-Means can pick up more important features rather than strictly pixel values. Similarly, results could be improved by reducing the dimensionality due to the curse of dimensionality causing the space between data points to increase rapidly as the number of features increases, thus making distance calculations (the basis of K-Means) less meaningful and more homogenized. The issue mentioned Section (5.3) with utilizing dimensionality reduction could be potentially be alleviated by using a pretrained dimensionality reduction model, or by finding another dimensionality reduction technique that can be trained incrementally with limited numbers of samples at a time. Even K-Means itself could be utilized as a form of dimensionality reduction, thus reutilizing the value of the distributed K-Means class, but reducing the colors in the image from RGB values to a mapping between single integers and RGB values found by clusters in K-Means.

Other improvements will likely come from optimizing the use of PySpark. This was a first attempt at a more detailed use of PySpark, and as such has limited optimization throughout the code in terms of both performance and storage optimizations. RDDs seemed to have largely become an outdated construct when looking for solutions to encountered problems, and PySpark Dataframes have fallen into favor. However, PySpark dataframes were not utilized in this project as they would have provided another barrier to entry, and were left instead as a potential improvement of the algorithm after more familiarity with the system and the complications mentioned in Section (5.3) were remedied.

5.2 Course Technologies and Skills

By implementing a distributed K-Means class, the requirements of this project appear to be satisfied. The project used a virtualization platform as its basis of implementation (Jetstream2 Exosphere virtual machines). The project utilized distributed processing via PySpark, includes some data cleaning in the sense of removing instances of the dataset that do not conform to the needed format (removing greyscale so RGB can be processed and converting RGBa to RGB for consistency), and also has published the full project to a public repository on GitHub, including the project code, links to the source data, a README file describing the project, data, and operation, and a metadata file sourced from the original data author's publication [4] detailing the data source, author, and description. Finally, the project meets the first option of a hands-on project, specifically the requirement of running an algorithm using a parallel programming framework by running K-Means using PySpark.

This project most directly connects to the `Processing and Analytics` module of the course. This module primarily handled utilizing distributed programming via PySpark to do basic analytics of text documents. This project then extends on this by going beyond text documents and implementing a data agnostic (in the sense that as just needs to be input as an indexed RDD where each instance utilizes the same numerical features) method of doing distributed clustering using Lloyd's Algorithm of K-Means clustering. The skills that this project helps to improve upon, as introduced by the course, are working with unstructured data (images are not all the same size and cannot easily fit into a relational database structure), utilizing virtualization and containerization by doing the analytics in a Docker container within a virtual machine, and working with distributed computing.

5.3 Barriers and Failures

There were some issues encountered while implementing the distributed K-Means class. First and foremost, there were issues with keeping the amount of occupied memory and disk storage down to usable levels. As the K-Means class would run, PySpark would store its needed temporary files on the disk, but once the program terminated naturally, these temporary files would not get removed. There were similar issues with PySpark storing files in memory, and not releasing the memory once the run was completed. Tinkering with various `unpersist` calls and garbage collect calls did not seem to have any effect whatsoever. This was largely an issue as it caused limitations with how large of a dataset could be processed at all, rather than just limiting how much of the dataset could be processed at any given time. Ultimately, this was worked around by simplifying the dataset and also manually clearing the temporary files between runs to free the disk storage and restarting the Python kernel to free the memory usage.

Another issue encountered when working on this project, was the limitations of using a dataset that was wide rather than long (large number of features rather than a large number of data instances). The initial plan was to first process the images by using an Incremental PCA approach to reduce the dimensionality, not only to make processing quicker (but still distributed) but also to improve the quality of the results, as compressing the images could help pick out more relevant features, rather than just a strict mapping to the average color of the image classes. This posed a problem, though. To do PCA, the full data would need to be loaded in to memory, or chunks of the data could be loaded in to memory by utilizing incremental PCA. However, the number of principal components that result from PCA is limited to the minimum of the number of instances and the number of features, which in this case would be the number of instances, especially with incremental PCA. This resulted in only being able to obtain around 100 to 200 principal components if the images were reduced in size enough, but this then doesn't account for a reasonable amount of the variance in the features of the dataset. Thus, because of the memory limitations (which are an assumed part of doing distributed programming to begin with), dimensionality reduction via PCA was not a viable option. Similarly, no other viable dimensionality reduction techniques were found such that they could be done in an incremental manner since the full dataset cannot fit into memory.

6 Conclusion

This project has successfully implemented a distributed K-Means class in Python that allows users to cluster data too large to fit within a single machine's memory by utilizing the distributed computing framework provided by PySpark. This distributed clustering class was then utilized to cluster the popular iris data set as well as a set of weather images provided by Kaggle, andcompared the results of the iris data clustering to the K-Means class provided by Sci-Kit Learn. The project was connected back primarily to the `Processing and Analytics` module of the course, and described how skills such as working with unstructured data, virtualization, and distributed computing were all utilized in putting together the distributed K-Means class.

7 References

[1] William McWhorter. info-i-535-project. `https://github.com/wimcwho/info-i-535-project`, 2024.

[2] Annette Catherine Paul. Introduction to Clustering Algorithms and its Use Cases — medium.com, 2021.

[3] Deepti Sisodia, Lokesh Singh, Sheetal Sisodia, and Khushboo Saxena. Clustering techniques: a brief survey of different clustering algorithms. *International Journal of Latest Trends in Engineering and Technology (IJLTET)*, 1(3):82–87, 2012.

[4] Haixia Xiao. Weather phenomenon database (WEAPD), 2021.

[5] Rui Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

A Distributed K-Means Implementation

```python
from pyspark import SparkContext, SparkConf
import random
import numpy as np
from tqdm import tqdm
```

```python
class KMeans:
    n_clusters = None
    cluster_centers = None
    inertia_ = None
    n_iter = None
    n_features_in = None
    min_iter = None
    max_iter = None
    n_init = None

    def __init__(self, n_clusters=2):
        assert n_clusters >= 1, 'number of clusters must be at least 1'

        self.n_clusters = n_clusters
        self.n_iter = 0
        self.n_features_in = 0

    def __dist(self, x, y):
        """calculates the euclidean distance between two numpy arrays"""

        return np.dot(x-y, x-y)

    def __closest_cluster(self, x):
        """calculates the closest cluster to a given numpy array"""
        distances = [self.__dist(x, cluster[1]) for cluster
                     in sorted(self.cluster_centers, key=lambda y: y[0])]
        return np.argmin(distances)

    def __inertia(self, rdd):
        """Calculate total inertia of KMeans fit using the training data
        rdd: RDD in form of (observation id, (observation value, cluster id))
             for each row"""

        clusters = sorted(self.cluster_centers, key=lambda x: x[0])
        distances = rdd.map(lambda x: self.__dist(clusters[x[1][1]][1], x[1][0]))
        val = distances.sum()
        distances = distances.unpersist()
        distances = None

        return val

    def predict(self, rdd):
        """Assign cluster IDs to observations in the RDD.
        rdd: RDD in form of (observation index, observation array) for each row"""

        assert self.cluster_centers is not None, 'must fit model first'

        rdd = rdd.map(lambda x: (x[0], self.__closest_cluster(x[1])))
        return rdd

    def fit(self, data, method='kmeans++', n_init=1, min_iter=0, max_iter=300):
        """Learn cluster centroids from data.
        data: RDD in form of (observation index, observation array) for each row
        n_iter: integer >= 1. indicates how many random initializations to be used,
                keeping the initialization with the lowest overall inertia.
        min_iter: minimum number of iterations to run before checking for
                  convergence. Can provide execution speed ups with larger cluster
                  sizes and to a lesser extent larger feature spaces
        max_iter: maximum number of iterations to run before stopping early."""

        assert n_init>=1, 'n_init must be an integer of at least 1'
        assert data.count() >= self.n_clusters, """number of clusters must be less
                                                  than or equal to number of
                                                  training instances"""
        assert method in ['kmeans++', 'kmeans'], """method must be
                                                   'kmeans++' or 'kmeans'"""

        self.n_init = n_init
        self.min_iter = min_iter
        self.max_iter = max_iter
        self.n_features_in = len(data.first()[1])
        self.__num_partitions = data.getNumPartitions()

        data = data.unpersist()

        self.__fit(data, min_iter, max_iter, method)
        SparkContext._jvm.System.gc()

        if n_init==1:
            return

        best_params = {'inertia':self.inertia_,
                       'centroids': self.cluster_centers,
                       'n_iter': self.n_iter}

        for i in range(n_init-1):
            self.__fit(data, min_iter, max_iter)
            SparkContext._jvm.System.gc()

            if self.inertia_ < best_params.get('inertia'):
                best_params['inertia'] = self.inertia_
                best_params['centroids'] = self.cluster_centers
                best_params['n_iter'] = self.n_iter

        self.inertia_ = best_params['inertia']
        self.cluster_centers = best_params['centroids']
        self.n_iter = best_params['n_iter']


    def __fit(self, data, min_iter, max_iter, method):
        """Learn cluster centroids from data. Only used as a sub method of fit(),
           but is equivalent to fit(data, n_init=1).
        data: RDD in form of (index, observation array) for each row
        min_iter: minimum number of iterations to run before checking for
                  convergence. Can provide execution speed ups with larger cluster
                  sizes and to a lesser extent larger feature spaces
        max_iter: maximum number of iterations to run before stopping early."""

        n_obs = data.count()

        # initial centers
        if method=='kmeans++':
            for k in tqdm(range(self.n_clusters)):
                if k==0:
                    rand_num = random.sample(range(n_obs), k=1)[0]
                    clusters = (data.filter(lambda x: x[0]==rand_num)
                                    .map(lambda x: x[1])
                                    .collect())
                else:
                    distances = data.map(lambda x:
                                    (x[0], min([self.__dist(x[1], cluster)
                                                for cluster in clusters])**2))
                    denom = distances.map(lambda x: x[1]).sum()
                    distances = distances.map(lambda x: (x[0], x[1]/denom))
                    sample_probs = (distances.sortByKey(ascending=True)
                                             .map(lambda x: x[1])
                                             .collect())
                    next_cluster_index = random.choices(range(n_obs),
                                                    weights=sample_probs,k=1)[0]
                    clusters.append(data.filter(lambda x: x[0]==next_cluster_index)
                                        .map(lambda x: x[1])
                                        .first())

            self.cluster_centers = list(zip(range(self.n_clusters), clusters))
            del clusters
        else:
            start = random.sample(range(n_obs), k=self.n_clusters)
            self.cluster_centers = (data.filter(lambda x: x[0] in start)
                                        .zipWithIndex()
                                        .map(lambda center:
                                             (center[1], center[0][1])))
            self.cluster_centers = self.cluster_centers.collect()

        #data -> (obs_id, (obs_val, center_id))
        data = data.map(lambda x: (x[0], (x[1], self.__closest_cluster(x[1]))))

        for i in tqdm(range(max_iter)):
        #for i in range(max_iter):
            #keep number of partitions from scaling up
            if data.getNumPartitions() > self.__num_partitions:
                data = data.repartition(self.__num_partitions)

            #number of observations in each cluster
            member_counts = (data.map(lambda x: (x[1][1], 1))
                                 .reduceByKey(lambda a,b: a+b)
                                 .collect())
            member_counts = sorted(member_counts, key=lambda x: x[0])

            #fix member_counts if any cluster has 0 members
            if len(member_counts) < self.n_clusters:
                present = [c_id for c_id, count in member_counts]
                for c_id in range(self.n_clusters):
                    if c_id not in present:
                        member_counts.append((c_id, 0))
                member_counts = sorted(member_counts, key=lambda x: x[0])

            #calculate locations of new centroids
            new_centers = (data.map(lambda x: (x[1][1], x[1][0]))
                               .reduceByKey(lambda x,y:
                                    x.astype('float32')+y.astype('float32'))
            new_centers = new_centers.map(lambda x:
                                          (x[0], x[1]/member_counts[x[0]][1]))

            #fix centroids if any cluster has 0 members
            num_centers = new_centers.count()
            #print(f'num centers: {num_centers}\nmember counts: {member_counts}')
            if num_centers < self.n_clusters:
                empty_clusters = [cid for cid, count in member_counts if count==0]
                rand_nums = random.sample(range(n_obs),
                                          k=self.n_clusters-num_centers)
                subset = (data.filter(lambda x: x[0] in rand_nums)
                              .zipWithIndex()
                              .map(lambda x: (empty_clusters[x[1]], x[0][1][0])))
                new_centers = (new_centers.union(subset)
                                          .coalesce(self.__num_partitions))

            #check for convergence and end fitting if convergence has be achieved
            old_centroids = [centroid[1] for centroid
                             in sorted(self.cluster_centers, key=lambda x: x[0])]
            new_centroids = np.array([centroid[1] for centroid
                                      in sorted(new_centers.collect(),
                                                key=lambda x: x[0])])
            if i > min_iter and np.isclose(old_centroids, new_centroids).all():
                #print('convergence achieved')
                self.n_iter = i
                break

            #update old centers and assign updated cluster
            #numbers to the data points
            self.cluster_centers = new_centers.collect()

            #data -> (obs_id, (obs_val, center_id))
            data = data.map(lambda x:
                            (x[0], (x[1][0],
                                    self.__closest_cluster(x[1][0])))).unpersist()

            new_centers = new_centers.unpersist()
            new_centers = None

        if self.n_iter == 0:
            self.n_iter = max_iter

        #find total inertia of the KMeans fit
        self.inertia_ = self.__inertia(data)

        #cleanup just in case
        try:
            subset = subset.unpersist()
        except:
            pass
        data = data.unpersist()
        new_centers = new_centers.unpersist()

        subset = None
        data = None
        new_centers = None

        SparkContext._jvm.System.gc()
```

B Image Preprocessing

B.1 Imports and Setup

```
from pyspark import SparkContext, SparkConf
import zipfile
import os
import shutil
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import seaborn as sns
from tqdm import tqdm
from itertools import permutations
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans as SKMeans

try:
    from cv2 import resize as imresize
except:
    !pip install opencv-python
    from cv2 import resize as imresize
```

```
IMAGE_DIR = './dataset/'

conf = SparkConf().set('spark.driver.memory', '3g')
conf = conf.set('spark.executor.memory', '1g')
conf = conf.set('spark.driver.maxResultSize', '2g')
sc = SparkContext(conf=conf)
```

B.2 Function Definitions

```
#Function Definitions
def RGBfloat2RGBint(arr):
    """convert an RGB image from float format to uint8 format"""

    assert (np.issubdtype(arr.dtype, np.floating) or
            np.issubdtype(arr.dtype, np.integer)), 'array is not a numerical dtype'

    if np.issubdtype(arr.dtype, np.floating):
        arr = 255*arr
        arr[arr<0]=0
        arr[arr>255]=255
        return arr.astype('uint8')[:,:,:3]
    else:
        return arr
```

```
def RGBa2RGB(arr):
    """convert an RGBa image to RGB by dropping the alpha channel"""

    return arr[:,:,:3]
```

```
def find_median(images):
    """calculate the median height and median width of all provided images.
    Images should be a PySpark RDD"""
    heights = (images.map(lambda image: image[1].shape[0])
                     .sortBy(lambda height: height)
                     .zipWithIndex()
    widths = (images.map(lambda image: image[1].shape[1])
                     .sortBy(lambda width: width)
                     .zipWithIndex()

    num_images = images.count()
    if num_images%2 == 0:
        lower_index = num_images//2
        median_height = np.mean(heights.filter(lambda height: height[1]
                                               in [lower_index, lower_index+1])
                                     .collect())
        median_width = np.mean(widths.filter(lambda width: width[1]
                                               in [lower_index, lower_index+1])
                                   .collect())
    else:
        median_height = (heights.filter(lambda height: height[1]==num_images//2)
                               .collect())[0]
        median_width = (widths.filter(lambda width: width[1]==num_images//2)
                             .collect())[0]

    median_height = int(np.round(median_height))
    median_width = int(np.round(median_width))

    heights = heights.unpersist()
    widths = widths.unpersist()

    return median_height, median_width
```

```
def find_mean(images):
    """calculate the mean height and mean width of all provided images.
    Images should be a PySpark RDD"""

    heights = (images.map(lambda image: image[1].shape[0])
                     .sortBy(lambda height: height))
    widths = (images.map(lambda image: image[1].shape[1])
                    .sortBy(lambda width: width))

    mean_height = int(heights.mean())
    mean_width = int(widths.mean())

    heights = heights.unpersist()
    widths = widths.unpersist()

    return mean_height, mean_width
```

```
def resize_images(images, size):
    """resize all images to be the provided size. Images should be a PySpark RDD"""

    if size=='median':
        height, width = find_median(images)
    elif size=='mean':
        height, width = find_mean(images)
        height = np.round(height)
        width = np.round(width)
    else:
        height, width = size

    dimensions = (height, width, 3)

    images = images.map(lambda image: (image[0], imresize(image[1],
                                                           (width, height))))

    return images, dimensions
```

B.3 Image Loading and Transformations

```
#if images are still in zip file, unzip and filter down to desired images,
#else move on
unique_labels = ['lightning', 'rain', 'rainbow', 'sandstorm', 'fogsmog']
if not os.path.isdir(IMAGE_DIR):
    #archive.zip is the zip file containing the images downloaded from Kaggle
    with zipfile.ZipFile('./archive.zip', 'r') as zip_ref:
        zip_ref.extractall('./')

    counter = 0
    for label in tqdm(os.listdir(IMAGE_DIR), total=len(os.listdir(IMAGE_DIR))):
        cur_dir = IMAGE_DIR+label+'/'

        #reduce disk storage load while first implementing
        if label not in unique_labels:
            shutil.rmtree(cur_dir)
            continue

        #rename images since dataset has duplicate names across labels
        for image_name in os.listdir(cur_dir):
            new_name = str(counter)

            #have all names be the same length
            if len(new_name) < 4:
                new_name = (4-len(new_name))*'0' + new_name
            new_name = 't' + new_name + '.jpg'

            os.rename(cur_dir+image_name, cur_dir+new_name)
            counter += 1

    del counter
    del label
    del cur_dir
    del image_name
    del new_name
```

```
#construct set of image name and class label pairs
image_labels = sc.emptyRDD()
for label in os.listdir(IMAGE_DIR):
    cur_dir = IMAGE_DIR+label+'/'
    image_labels = (image_labels.union(sc.parallelize(os.listdir(cur_dir))
                                .map(lambda k: (k, label))))

del label
del cur_dir
```

```
#import images as arrays
```

```
images = image_labels.map(lambda image:
                          (image[0], mpimg.imread(IMAGE_DIR+image[1]+'/'+image[0])))
```

```
#remove images that aren't in rgb format
not_rgb_images = (images.filter(lambda image: len(image[1].shape) < 3)
                        .keys()
                        .collect())

images = images.filter(lambda image: image[0] not in not_rgb_images)
image_labels = image_labels.filter(lambda image: image[0] not in not_rgb_images)

num_images = images.count()
images = images.unpersist()
```

```
#change images to be integer formatted RGB instead of some being float formatted
images = images.map(lambda image: (image[0], RGBa2RGB(RGBfloat2RGBint(image[1]))))
```

```
#resize the images
images, dimensions = resize_images(images, 'mean')
```

```
#flatten the image arrays
images = images.map(lambda image: (image[0], image[1].reshape(-1)))
```

```
#convert to a format required by the distirbuted K-Means class
images = images.zipWithIndex()
image_names = images.map(lambda x: (x[1], x[0][0])) # (image id, image name)
images = images.map(lambda x: (x[1], x[0][1])) # (image id, image array)
```