Dependencies

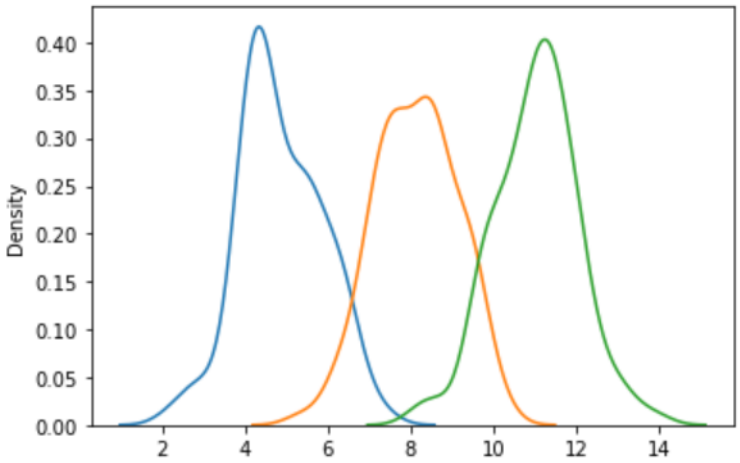
- numpy
- sklearn.mixture.GaussianMixture
- scipy.stats.kstest
- cv2

Generating Synthetic Data

We begin by selecting three pre-determined centroids for 3-means clustering. I selected 5, 8, and 11 as pre-determined means for the purposes of this experiment.

We can now generate three high dimensional vectors of size 100 from the Gaussian normal distribution with the predetermined means and same variance (=1) as follows:

```
# Pre-determined means for clustering
means = np.array([5,8,11])
data = []

for i in means:
    # Generating k=3 high dimensional vectors
    vec = np.random.normal(loc=i, scale=1, size=100)
    data.append(vec)
    # Plot distribution of each high dimensional vector on same plot
    sns.kdeplot(vec)
```



In the figure above, we notice that the high dimensional vectors have overlapping elements. These areas of overlap are where our same-cluster-queries algorithm will improve clustering.

The next step is to generate Gaussian normal variables by a Gaussian mixture model. To perform this step, I used the GaussianMixture scikit-learn package. Using our high dimensional Gaussian normal vectors as mean vectors, we initialise our model with 3 components and a shared, fixed covariance matrix. After fitting the model to our mean vector matrix, we use it to sample 300 vectors of size 100.

Generate Gaussian normal variables by a Gaussian mixture model

- https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html#sklearn.mixture.GaussianMixture.fit

```
# Initialize GaussianMixture model with 3 centroids, randomly initialize with one shared covariance matrix
gmm = GaussianMixture(n_components=3, init_params='random', covariance_type='tied', random_state=0)
gmm.fit(data)
```

[3]:
```
GaussianMixture(covariance_type='tied', init_params='random', n_components=3,
                random_state=0)
```

```
# Sample 300 vectors of size 100 from fitted GaussianMixture Model
vecs = gmm.sample(300)[0]
vecs.shape
```

[4]:
```
(300, 100)
```

These 300 vectors are inputs to our clustering algorithms. We can use them to judge algorithm performance.

Lloyd's Algorithm

With our synthetic data prepared, we can implement 3-means clustering using Lloyd's algorithm. The following algorithm has been adopted from https://datasciencelab.wordpress.com/2013/12/12/clustering-with-k-means-in-python/

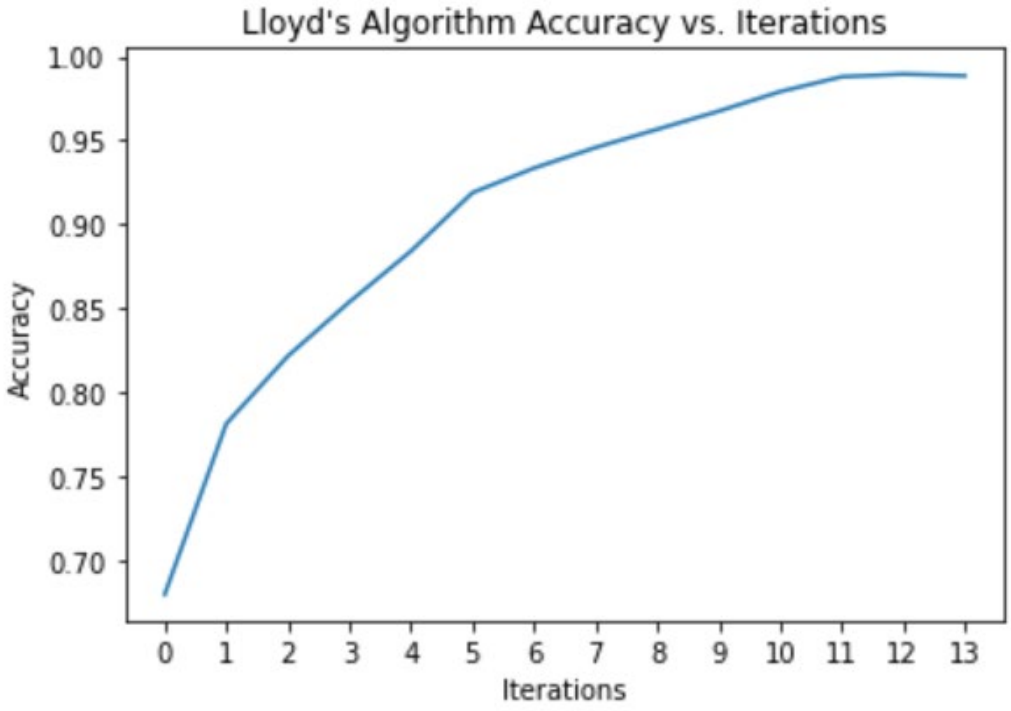
It has been modified to include accuracy calculations after every iteration when the centroids are re-evaluated.

```
def cluster_points(X, mu):
    clusters  = {}
    for x in X:
        bestmukey = min([(i[0], np.linalg.norm(x-mu[i[0]])) \
                    for i in enumerate(mu)], key=lambda t:t[1])[0]
        try:
            clusters[bestmukey].append(x)
        except KeyError:
            clusters[bestmukey] = [x]
    return clusters
```

```
def reevaluate_centers(mu, clusters):
    newmu = []
    keys = sorted(clusters.keys())
    for k in keys:
        newmu.append(np.mean(clusters[k], axis = 0))
    return newmu

def has_converged(mu, oldmu):
    return (set([tuple(a) for a in mu]) == set([tuple(a) for a in oldmu]))

def kmeans(X, K, seed=42):
    # Plant seed to track random samples
    random.seed(seed)
    # Initialize to K random centers
    oldmu = random.sample(X, K)
    mu = random.sample(X, K)
    # Use to track iterations vs. accuracy
    accuracy_dict = {}
    count = 0
    while not has_converged(mu, oldmu):
        oldmu = mu
        # Assign all points in X to clusters
        clusters = cluster_points(X, mu)
        # Calculate accuracy
        mu_means = np.array(pd.Series(mu).apply(lambda x: x.mean()))
        # Sort means in ascending order
        mu_means.sort()
        # Accuracy = 1- average error of means
        accuracy_dict[count] = 1 - np.average(np.abs(np.array(mu_means) - np
        print('Iteration {}: {}'.format(count, mu_means))
        # Increase number of iterations by 1
        count += 1
        # Reevaluate centers
        mu = reevaluate_centers(oldmu, clusters)
    return (mu, clusters), accuracy_dict
```

Since we already know the ideal centroids from the predetermined means (5, 8, 11), we can calculate the accuracy after each iteration of Lloyd's algorithm. Accuracy is calculated as 1 – average error of centroids.

Plotting the accuracy vs. iterations in a graph below:



Lloyd's algorithm converges very closely towards the predetermined means. The final error in the range 1e-02 can be dismissed as random noise occurring from having finite vectors of size 100.

Clustering with Same Cluster Querying

Although Lloyd's algorithm provides a good foundation for k-means clustering, it takes a lot of iterations to converge to the ideal centroids. In this section of the paper, I aim to improve the performance of Lloyd's algorithm using same cluster querying.

Kolmogorov-Smirnov Test

The algorithm I have developed uses the scipy kstest statistic to determine whether two vectors come from the same distribution.

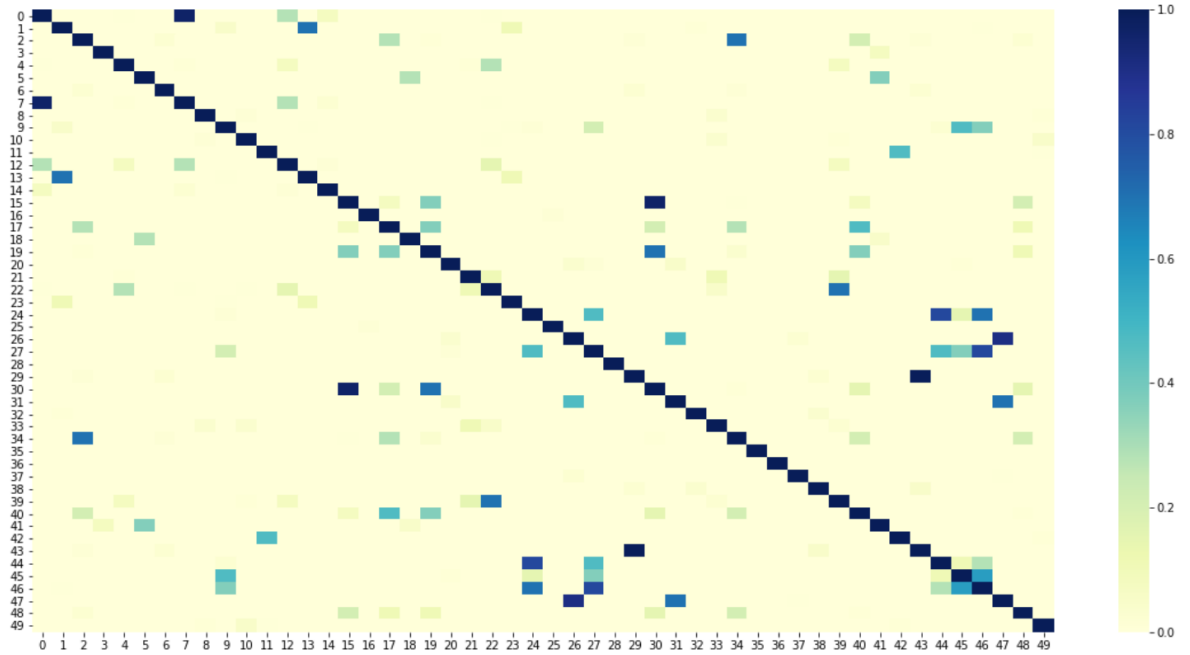
"This test performs the Kolmogorov-Smirnov test for goodness of fit. The two-sample test tests whether the two independent samples are drawn from the same continuous distribution. Under the null hypothesis, the two distributions are identical."

The scipy kstest statistic also returns a p-value for each test which will help us make a correct clustering determination based on a fixed significance level. To visualize this statistic over our sample vectors, I generated a kstest p-value matrix from the first 50 vectors and plotted a heatmap.

```
ks_matrix = np.zeros(shape=(50,50))
for i in range(len(vecs[:50])):
    for j in range(len(vecs[:50])):
        ks_matrix[i][j] = stats.kstest(vecs[i], vecs[j]).pvalue
```

```
fig_dims = (20, 10)
fig, ax = plt.subplots(figsize=fig_dims)
sns.heatmap(ks_matrix, cmap="YlGnBu", ax=ax)
```

```
<AxesSubplot:>
```



Observe that the vectors that are guaranteed to come from the same distribution along the diagonal have p-value = 1. Vectors with p-value ≈ 0 are guaranteed to come from different distributions. It follows that p-values are positively correlated with correct clustering of Gaussian normal vectors.

Same cluster querying algorithm

This algorithm works by re-evaluating clusters in each iteration of Lloyd's algorithm. After vectors are clustered to the closest centroid, we calculate the kstest statistic (p-value) of all vectors with average cluster vectors for each cluster. We then determine if the vector belongs to the same distribution as its assigned cluster by setting a significance level. By default, we set the significance level to 0.95.

That is, if any vector has a p-value lower than 0.95 with its average cluster vector, we assume it has been erroneously clustered. We correct errors arising by re-clustering the vectors to those clusters with whose average cluster vector they have the highest kstest statistic (pvalue). Therefore, fewer iterations are required for convergence.

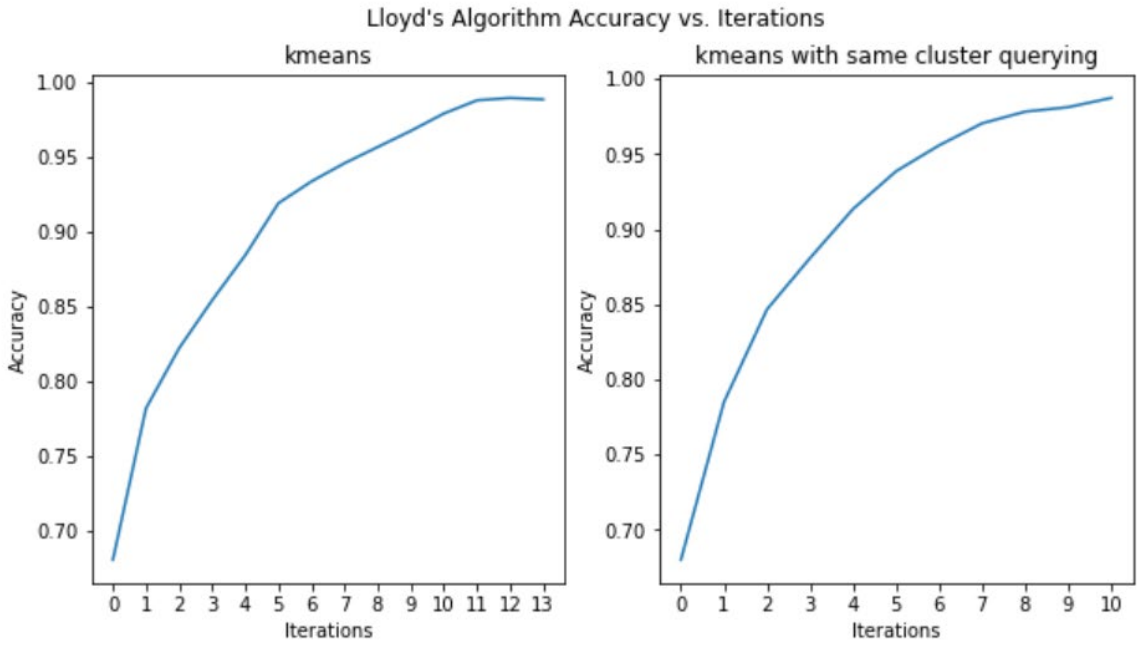
```python
def reevaluate_clusters(X, clusters, tol):
    '''
    Function to correct clustering errors from Lloyd's algorithm.

    Calculate the kstest statistic (pvalue) of all vectors with
    their average cluster vector to determine if the vector belongs
    to the same distribution as the rest of the cluster.

    Correct errors arising in the kmeans clustering by all vectors to
    clusters with whose average cluster vector they have the
    highest kstest statistic (pvalue).
    '''
    avg_vecs = {}
    pvals = {}
    removed = []

    # Store average cluster vectors
    for key in clusters.keys():
        avg_vecs[key] = sum(clusters[key]) / len(clusters[key])

    # Remove poorly clusterd vector from cluster
    for key in clusters.keys():
        for x in X:
            # If kstest pvalue lower than tolerance, vector is poorly clustered
            stat = stats.kstest(x, avg_vecs[key]).pvalue
            if stat < tol:
                try:
                    clusters[key].remove(x)
                    # Store removed vectors to be re-clustered
                    removed.append(x)
                except ValueError:
                    pass

    # Re-distribute removed vectors to the cluster with max kstest pvalue
    for vec in removed:
        for key in clusters.keys():
            # Store pvalue of kstest stat for each vector
            pvals[key] = stats.kstest(vec, avg_vecs[key]).pvalue

        temp = max(pvals.values())
        key = [key for key in pvals if pvals[key] == temp]
        clusters[key[0]].append(vec)

    # Return re-evaluated clusters
    return clusters
```

The updated k-means algorithm is shown below:

CLUSTERING WITH SAME CLUSTER QUERIES

```
def kmeans_new(X, K, tol=0.95, seed=42):
    # Plant seed to track random samples
    random.seed(seed)
    # Initialize to K random centers
    oldmu = random.sample(X, K)
    mu = random.sample(X, K)
    # Use to track iterations vs. accuracy
    accuracy_dict = {}
    count = 0
    while not has_converged(mu, oldmu):
        oldmu = mu
        # Assign all points in X to clusters
        clusters = cluster_points(X, mu)
        # Re-evaluate clusters
        clusters = reevaluate_clusters(X, clusters, tol)
        # Calculate accuracy
        mu_means = np.array(pd.Series(mu).apply(lambda x: x.mean()))
        # Sort means in ascending order
        mu_means.sort()
        # Accuracy = 1- average error of means
        accuracy_dict[count] = 1 - np.average(np.abs(np.array(mu_means) - np.array(means)) / means)
        # If maximum accuracy is achieved, stop iterating
        if accuracy_dict[count] == 1:
            break
        print('Iteration {}: {}'.format(count, mu_means))
        # Increase number of iterations by 1
        count += 1
        # Reevaluate centers
        mu = reevaluate_centers(oldmu, clusters)
    return (mu, clusters), accuracy_dict
```

Preliminary side-by-side performance comparison against Lloyd's algorithm:



It appears the updated k-means algorithm using same cluster queries achieves the same accuracy in fewer iterations. However, the time complexity of the updated kmeans algorithm is larger than Lloyd's algorithm as clusters must be re-evaluated in each iteration, a task that takes at least linear time. To better understand the time-complexity trade-offs, we conduct further testing.

Time-Complexity Testing

To further test and compare the two algorithms, we perform 100 iterations of 3-means clustering with different random seeds to ensure different clustering initialization points. We take note of the average runtime, average accuracy, and average iterations for each algorithm.

```
kmeans_i = []
kmeans_a = []
kmeans_time = []

kmeans_sc_i = []
kmeans_sc_a = []
kmeans_sc_time = []

# Test kmeans vs. kmeans_new 100 times with different random seeds for different random initializations
for i in range(100):
    t0 = time.time()
    accuracy_kmeans = kmeans(list(vecs), 3, seed=i)[1]
    t1 = time.time()
    kmeans_time.append(t1-t0)
    kmeans_i.append(np.max(list(accuracy_kmeans.keys())))
    kmeans_a.append(np.max(list(accuracy_kmeans.values())))

    t2 = time.time()
    accuracy_kmeans_new = kmeans_new(list(vecs), 3, seed=i)[1]
    t3 = time.time()
    kmeans_sc_time.append(t3-t2)
    kmeans_sc_i.append(np.max(list(accuracy_kmeans_new.keys())))
    kmeans_sc_a.append(np.max(list(accuracy_kmeans_new.values())))
```

...

```
'kmeans mean iterations: {}, \
kmeans mean accuracy: {}, \
kmeans mean time: {}'.format(np.average(kmeans_i), np.average(kmeans_a), np.average(kmeans_time))
```

```
'kmeans mean iterations: 9.99, kmeans mean accuracy: 0.9867889181749974, kmeans mean time: 0.09494743585586547'
```

```
'kmeans_new mean iterations: {}, \
kmeans_new mean accuracy: {}, \
kmeans_new mean time: {}'.format(np.average(kmeans_sc_i), np.average(kmeans_sc_a), np.average(kmeans_sc_time))
```

```
'kmeans_new mean iterations: 6.83, kmeans_new mean accuracy: 0.9791241563280831, kmeans_new mean time: 1.5284472274780274'
```

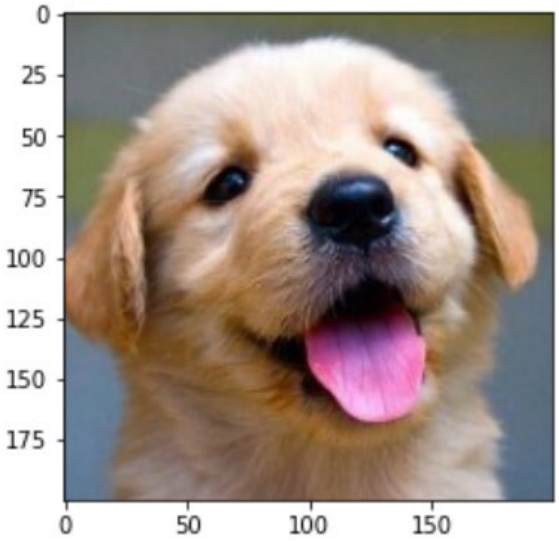
The updated algorithm takes approximately 3 iterations less on average to converge than Lloyd's algorithm, while maintaining the same average accuracy. However, one must note the increase in average runtime. This is to be expected as re-evaluating clusters takes approximately linear time with each iteration. Although relatively larger than Lloyd's algorithm average runtime, performing 3-means clustering over 300 high dimensional vectors still takes only 1.5 seconds with the updated algorithm.

Image Segmentation using Updated K-means

In this section, I aim to perform k-means clustering on 200x200 images using the updated algorithm. We begin by reshaping the images to 2D arrays of pixels and 3 colour values (RGB).

```
image = cv2.imread("image.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
plt.imshow(image)
```

```
<matplotlib.image.AxesImage at 0x2707e0d5108>
```

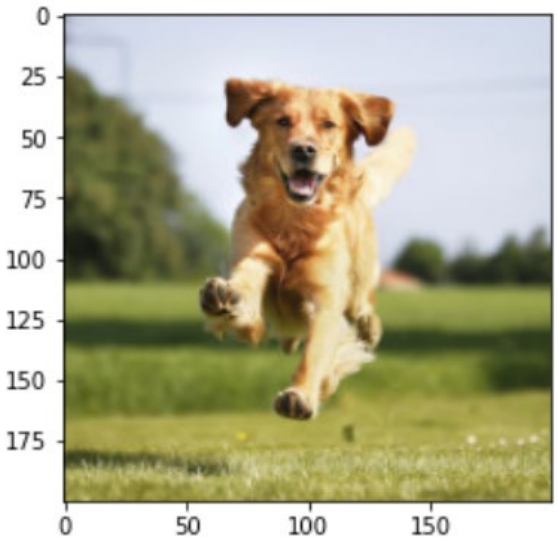



```
# reshape the image to a 2D array of pixels and 3 color values (RGB)
pixel_values = image.reshape((-1, 3))
# convert to float
pixel_values = np.float32(pixel_values)
print(pixel_values.shape)
```

```
(40000, 3)
```

```
image = cv2.imread("image.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
plt.imshow(image)
```

```
<matplotlib.image.AxesImage at 0x253e4262bc8>
```




```
# reshape the image to a 2D array of pixels and 3 color values (RGB)
pixel_values = image.reshape((-1, 3))
# convert to float
pixel_values = np.float32(pixel_values)
print(pixel_values.shape)
```

```
(40000, 3)
```

We can now run the updated k-means algorithm with 3 centres for image segmentation. However, we must first update our k-means algorithm to return labels for each vector and centres for each cluster. As the algorithm is not optimized for time-complexity, we expect a longer run-time. After 41 iterations, the final image returned can be seen below.

CLUSTERING WITH SAME CLUSTER QUERIES