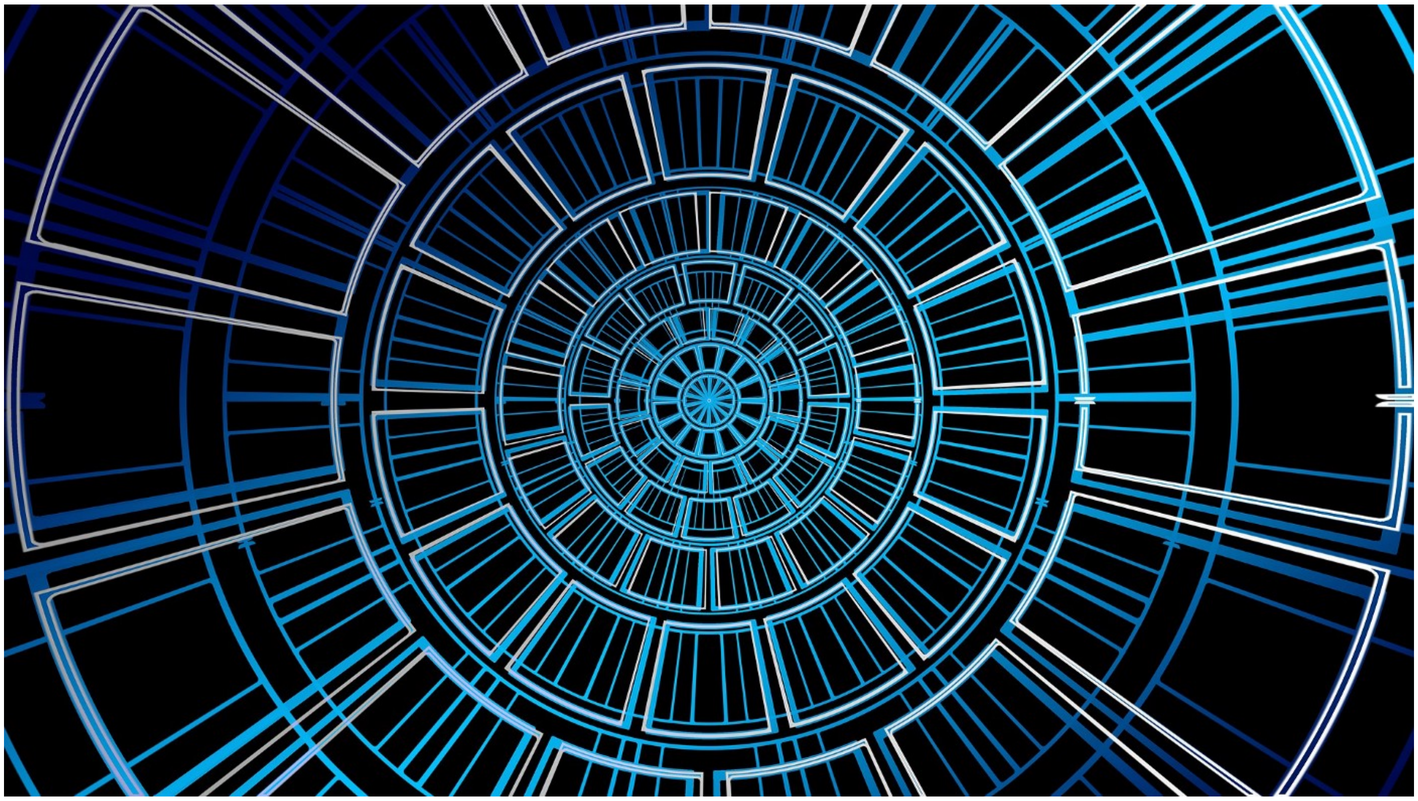
Daniel Foley Follow
Feb 8, 2019 · 12 min read ★ · Listen



Source: Pixabay

K-Means Clustering

Making Sense of Text Data using Unsupervised Learning

Customer Segmentation, Document Classification, House Price Estimation, and Fraud Detection. These are just some of the real world applications of clustering. There are many other use cases for this algorithm but today we are going to apply K-means to text data. In particular, we are going to implement the algorithm from scratch and apply it to the Enron email data set and show how this technique can be a very useful way of summarizing large amounts of text and uncovering useful insights that might otherwise not be feasible.

So what exactly is K-means? Well, it is an unsupervised learning algorithm (meaning there are no target labels) that allows you to identify similar groups or clusters of data points within your data. To see why it might be useful, imagine one of the use cases mentioned above, Customer Segmentation. A company using this algorithm would be able to partition their customers into different groups depending on their characteristics. This can be a very useful

While working as an Economist I was able to use this technique to analyze a public consultation where a lot of the responses were qualitative in nature. Making use of my machine learning knowledge I was able to create useful insights and get a feel for the data while avoiding quite a bit of manual work for my colleagues which went down quite well.

More Formally

Again the problem of K means can be thought of as grouping the data into K clusters where assignment to the clusters is based on some similarity or distance measure to a centroid (more on this later). So how do we do this? Well, let's first outline the steps involved.

1. ***We randomly initialize the K starting centroids. Each data point is assigned to its nearest centroid.***
2. ***The centroids are recomputed as the mean of the data points assigned to the respective cluster.***
3. ***Repeat steps 1 and 2 until we trigger our stopping criteria.***

Now you may be wondering what we are optimizing for and the answer is usually Euclidean distance or squared Euclidean distance to be more precise. Data points are assigned to the cluster closest to them or in other words the cluster which minimizes this squared distance. We can write this more formally as:

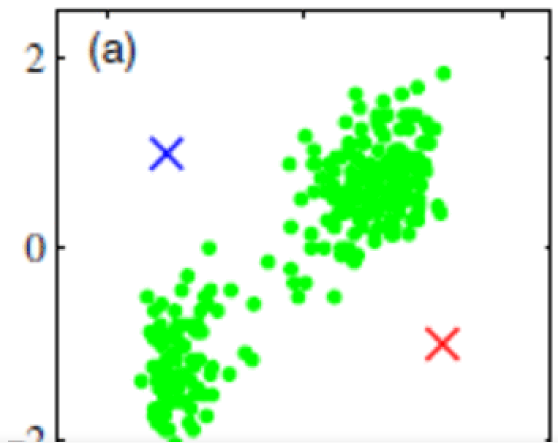
$$J = \Sigma_{n=1}^{N} \Sigma_{k=1}^{K} r_{nk} \left|\left| x_n - \mu_k \right|\right|^2$$

K means Cost Function

J is just the sum of squared distances of each data point to it's assigned cluster. Where ***r*** is an indicator function equal to 1 if the data point (x_n) is assigned to the cluster (k) and 0 otherwise. This is a pretty simple algorithm, right? Don't worry if it isn't completely clear yet. Once we visualize and code it up it should be easier to follow.

K means Visualised

I have always been a fan of using visual aids to explain topics and it has usually helped me gain a deeper intuition of what is actually happening with various algorithms. So let's see what K means looks like after each iteration.



As you can see the figure above shows K means at work. We have defined k = 2 so we are assigning data to one of two clusters at each iteration. Figure (a) corresponds to the randomly initializing the centroids. In (b) we assign the data points to their closest cluster and in Figure c we assign new centroids as the average of the data in each cluster. This continues until we reach our stopping criteria (minimize our cost function J or for a predefined number of iterations). Hopefully, the explanation above coupled with the visualization has given you a good understanding of what K means is doing. Next up, we are going to implement this algorithm in ***Python.***

Data set and Code

As I mentioned before, we are going to be using text data and in particular, we will be taking a look at the Enron email data set which is available on Kaggle. For those of you that don't know the story/scandal surrounding Enron, I would suggest checking out the ***smartest guys in the room***. It is a particularly good documentary on the subject.

Just One Issue

Can we just give our algorithm a bunch of text data and expect anything to happen? Unfortunately, no we can't. Algorithms have a hard time understanding text data so we need to transform the data into something the model can understand. Computers are exceptionally good at understanding numbers so how about we try that. If we represent the text in each email as a vector of numbers then our algorithm will be able to understand this and proceed accordingly. What we will be doing is transforming the text in the body of each email into a vector of numbers using **Term Frequency-Inverse Document Frequency or TF-IDF.** I won't go into too much detail on what this is as I have explained it in a previous post but essentially it enables us to calculate the importance of words in each email relative to what is in that email but also relative to all the emails in the data set. More info on TF-IDF is available here.

Ok so the first thing we need to do is import the required libraries and the data set. I should mention that I tried to use the full data set in a Kaggle kernel which was a bit of a challenge. I ran into some kernel failures so I ended up using about 70 per cent of the full data set which ran without any problems. Please note that I have not put the data cleaning portion of the code in the post since we are focusing on the K means algorithm. For those interested, the full code is available on my Kaggle Kernel linked at the end of the post. As always, we import the libraries we will be using and also read in the data set.

```python
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from sklearn.cluster import KMeans
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import PCA
from sklearn.preprocessing import normalize
from sklearn.metrics import pairwise_distances

import nltk
import string

```

```
# email module has some useful functions
import os, sys, email,re

df = pd.read_csv('../input/emails.csv',nrows = 35000)
```

imports.py hosted with ❤ by GitHub view raw

After we do a little bit of text cleaning, i.e. convert to lower case, remove stop words and HTML we can move on to using TF-IDF which is pretty straightforward to do in sklearn.

```
from sklearn.feature_extraction.text import TfidfVectorizer
data = df['body_new']


tf_idf_vectorizor = TfidfVectorizer(stop_words = 'english',#tokenizer = tokenize_and_stem,
                             max_features = 20000)
tf_idf = tf_idf_vectorizor.fit_transform(data)
tf_idf_norm = normalize(tf_idf)
tf_idf_array = tf_idf_norm.toarray()
```

tfidf.py hosted with ❤ by GitHub view raw

After running this code we can have a sneak peek at our feature names using the get_feature_names() method below.

```
pd.DataFrame(tf_idf_array, columns=tf_idf_vectorizor.get_feature_names()).head()
```

Now we need to think about what our K means class will look like. Well, there are a few methods that we need to implement and these correspond to the steps I outlined above. We will implement the following 5 methods which will help us split up the algorithm into manageable parts.

- ***Initialise_centroids***
- ***assign_clusters***
- ***update_centroids***

```python
class Kmeans:
    """ K Means Clustering

    Parameters
    -----------
        k: int , number of clusters

        seed: int, will be randomly set if None

        max_iter: int, number of iterations to run algorithm, default: 200

    Attributes
    -----------
       centroids: array, k, number_features

       cluster_labels: label for each data point

    """

    def __init__(self, k, seed = None, max_iter = 200):
        self.k = k
        self.seed = seed
        if self.seed is not None:
            np.random.seed(self.seed)
        self.max_iter = max_iter



    def initialise_centroids(self, data):
        """Randomly Initialise Centroids

        Parameters
        ----------
        data: array or matrix, number_rows, number_features

        Returns
        --------
        centroids: array of k centroids chosen as random data points
        """

        initial_centroids = np.random.permutation(data.shape[0])[:self.k]
        self.centroids = data[initial_centroids]

        return self.centroids
```

kmean1.py hosted with ❤ by GitHub view raw

The code above defines our Kmeans class, the init and the initialise_centroids methods. We want our class to take in some parameters such as the number of clusters, the number of iterations and the seed which we need for reproducibility. Setting the seed is an important step since we randomly initialize our centroids at the start of the algorithm. If we didn't set our seed then we may converge to a different set of clusters each time we ran the algorithm. The initialise_centroids method simply selects k random data points and sets them as the initial cluster centres to begin the algorithm.

We now need to write the methods for assigning data points to particular clusters and also to update the cluster centres. Remember we assign data to clusters depending on the Euclidean distance to the centre cluster. We use the **pairwise distance** method from sklearn which simplifies this calculation for us and returns the distances to each cluster centre. The argmin function identifies the index with the minimum distance to each cluster allowing us to assign the correct cluster label to that index. Now to finish off one iteration of the algorithm we just need to update the centroids as the average of all the data points assigned to the specific cluster.

```
        def assign_clusters(self, data):
          """Compute distance of data from clusters and assign data point
             to closest cluster.

          Parameters
          ----------
          data: array or matrix, number_rows, number_features

          Returns
          --------
          cluster_labels: index which minmises the distance of data to each
          cluster

          """

          if data.ndim == 1:
              data = data.reshape(-1, 1)

          dist_to_centroid =  pairwise_distances(data, self.centroids, metric = 'euclidean')
          self.cluster_labels = np.argmin(dist_to_centroid, axis = 1)

          return  self.cluster_labels


      def update_centroids(self, data):
          """Computes average of all data points in cluster and
             assigns new centroids as average of data points

          Parameters
          -----------
        """

        self.centroids = np.array([data[self.cluster_labels == i].mean(axis = 0) for i in range(self.k)])

        return self.centroids
```

kmeans2.py hosted with ❤ by GitHub view raw

The next two methods are also very important. The **predict** method basically returns the corresponding predicted cluster label for each data point based on our algorithm. The last method in the code snippet below fits our model by calling the functions we previously defined. Ok, that's pretty much it for our k-means class. Now we just have to figure out the optimal number of clusters to choose when running our algorithm.

```

    def predict(self, data):
        """Predict which cluster data point belongs to

        Parameters
        ----------
        data: array or matrix, number_rows, number_features

        Returns
        --------
        cluster_labels: index which minmises the distance of data to each
        cluster
        """

        return self.assign_clusters(data)

    def fit_kmeans(self, data):
        """
        This function contains the main loop to fit the algorithm
        Implements initialise centroids and update_centroids
        according to max_iter
        -----------------------

        Returns

        # Main kmeans loop
        for iter in range(self.max_iter):

            self.cluster_labels = self.assign_clusters(data)
            self.centroids = self.update_centroids(data)
            if iter % 100 == 0:
                print("Running Model Iteration %d " %iter)
        print("Model finished running")
        return self
```

kmeans3.py hosted with ❤ by GitHub view raw

Optimal Number of Clusters

When using K-means, one of the things we need to do is make sure we choose the optimal number of clusters. Too little and we could be grouping data together that have significant differences. Too many clusters and we will just be overfitting the data and our results will not generalise well. To answer this question we will use the ***elbow method*** which is a common technique used for this task. It involves estimating the model using various numbers of clusters and calculating the negative of the ***within-cluster sum of squares*** for each number of clusters chosen using the score method from sklearn. Notice that this is just the negative of our objective function above. We choose the number where adding further clusters only marginally increases the score. The result when graphed looks distinctly like an elbow (or upside down elbow in this case). The best choice for the number of clusters is where the elbow forms, 3 in our case and we can see this from the figure below. (We could also probably experiment with 4 clusters but for this implementation, we will go with 3)

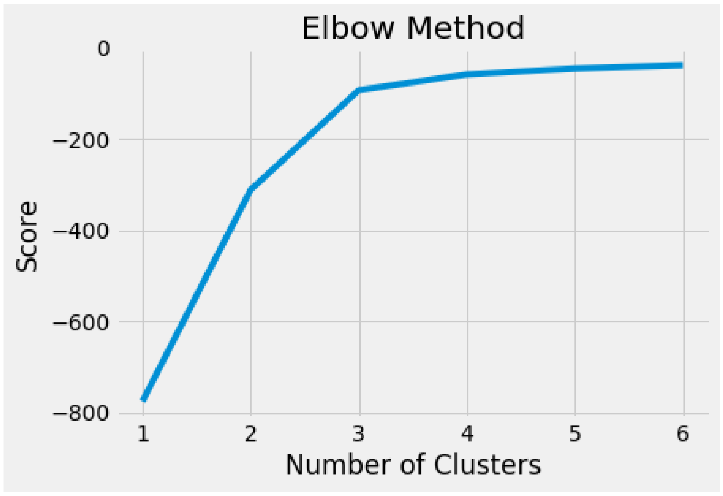
```
number_clusters = range(1, 7)

kmeans = [KMeans(n_clusters=i, max_iter = 600) for i in number_clusters]
kmeans

score = [kmeans[i].fit(Y_sklearn).score(Y_sklearn) for i in range(len(kmeans))]
score

```

elbow.py hosted with ❤ by GitHub view raw



Optimal Number of Clusters

My Implementation

In this part of the post, we are going to implement the algorithm we have just coded up in Python. In order to see our clusters graphically, we are going to use ***PCA*** to reduce the dimensionality of our feature matrix so we can plot it in two dimensions. With that said, we choose two components and transform our tf_idf_array using the fit_transform() method of the PCA class. Then we create an instance of our Kmeans class choosing 3 clusters prompted by our analysis above. Now it is just a case of calling the fit_kmeans() and predict() methods to put our data points into clusters. Since we have projected our array into a 2-d space we can easily use a scatter plot to visualize this along with the cluster centres.

```
sklearn_pca = PCA(n_components = 2)
Y_sklearn = sklearn_pca.fit_transform(tf_idf_array)
test_e = Kmeans(3, 1, 600)
fitted = test_e.fit_kmeans(Y_sklearn)
predicted_values = test_e.predict(Y_sklearn)

plt.scatter(Y_sklearn[:, 0], Y_sklearn[:, 1], c=predicted_values, s=50, cmap='viridis')

centers = fitted.centroids
plt.scatter(centers[:, 0], centers[:, 1],c='black', s=300, alpha=0.6);
```

kmeans_fit.py hosted with ❤ by GitHub view raw

We can see three pretty distinct clusters here with particularly large separation for the purple cluster indicating quite a difference in terms of the content of the emails. The majority of the data is contained within the green cluster, however.

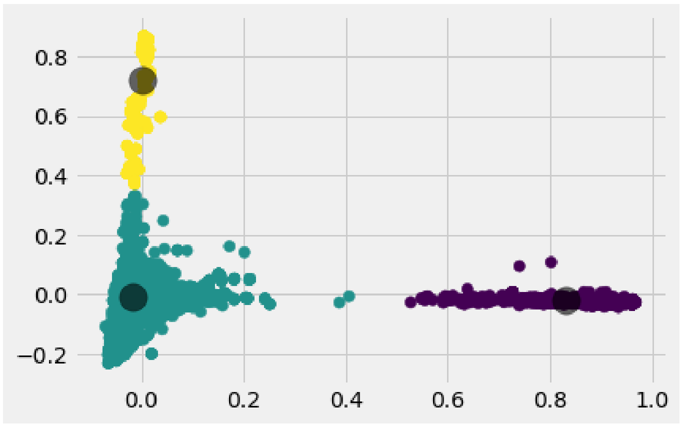


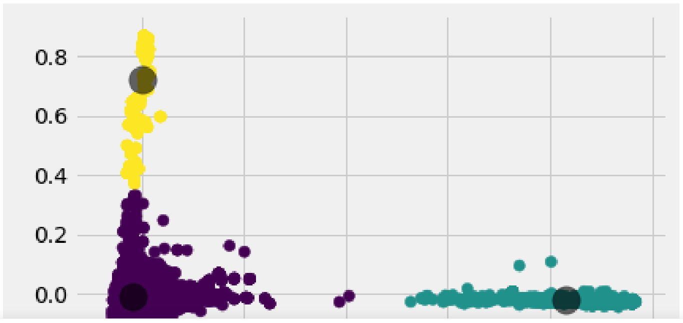
Figure 2: My Implementation Clustering Results

SK-Learn Implementation

Just as a sense check we are going to re-do this estimation using sklearn. In the real world, I highly doubt you would be implementing this from scratch as it just isn't really required. It is, however, a really useful way of concretely understanding how k-means works so definitely well worth doing yourself. We can see that sklearn makes the estimation much simpler and if we plot the results the two graphs look very similar. This is reassuring and makes it less likely that our own code has bugs in it. Although the colours of the clusters have swapped around for some reason??

```
from sklearn.cluster import KMeans
sklearn_pca = PCA(n_components = 2)
Y_sklearn = sklearn_pca.fit_transform(tf_idf_array)
kmeans = KMeans(n_clusters=3, max_iter=600, algorithm = 'auto')
fitted = kmeans.fit(Y_sklearn)
prediction = kmeans.predict(Y_sklearn)
```

sklearn.py hosted with ❤ by GitHub view raw



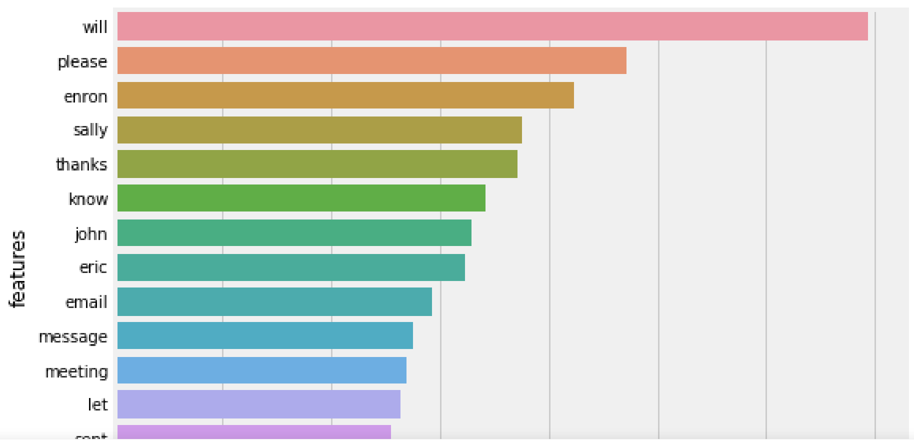
Examining the Top words in each Cluster

In this section, we are going to take a quick look at the results we got. What we are mainly interested in is seeing if there are any commonalities between words in each cluster or any particular words that stand out. In other words, can we identify themes in each cluster? If we can then this is a pretty powerful way of getting a general feel for what the emails contain and can guide any further analysis we wish to do and the best part is we didn't have to read 35,000 emails. We can view the top words in each cluster using the method below which just identifies the features with the highest mean tf_idf scores across each cluster.

```python
def get_top_features_cluster(tf_idf_array, prediction, n_feats):
    labels = np.unique(prediction)
    dfs = []
    for label in labels:
        id_temp = np.where(prediction==label) # indices for each cluster
        x_means = np.mean(tf_idf_array[id_temp], axis = 0) # returns average score across cluster
        sorted_means = np.argsort(x_means)[::-1][:n_feats] # indices with top 20 scores
        features = tf_idf_vectorizor.get_feature_names()
        best_features = [(features[i], x_means[i]) for i in sorted_means]
        df = pd.DataFrame(best_features, columns = ['features', 'score'])
        dfs.append(df)
    return dfs
dfs = get_top_features_cluster(tf_idf_array, prediction, 15)
```

top_features.py hosted with ❤ by GitHub view raw

Below are three graphs corresponding to the top 15 words in each cluster ordered by relative importance as measured by TF-IDF.



Open in app Get started

Figure 3: Cluster 0

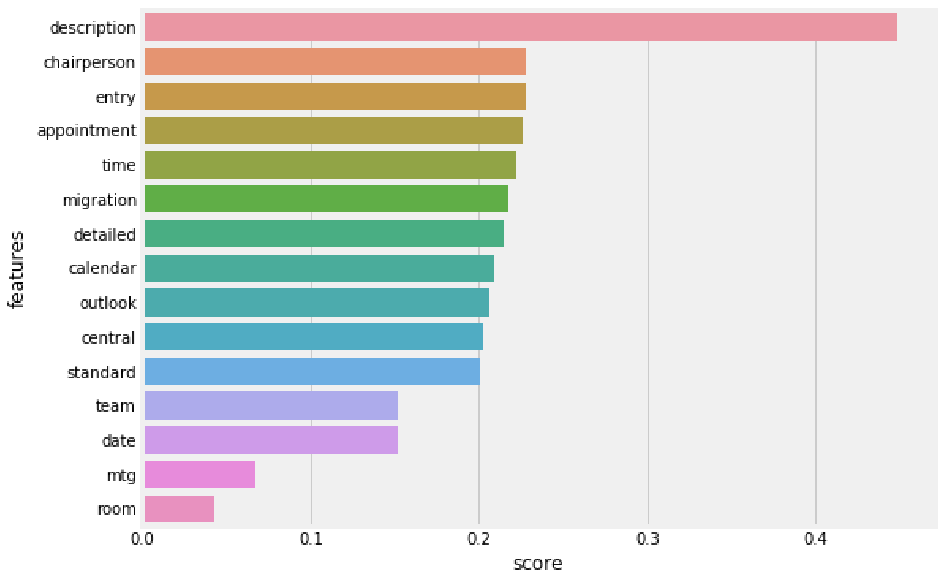


Figure 4: Cluster 1

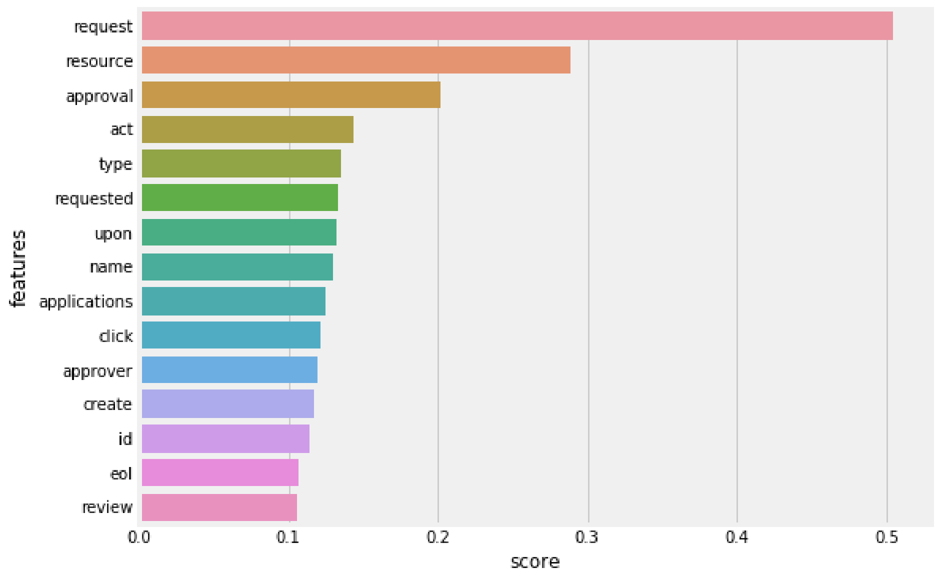


Figure 5: Cluster 2

Ok, so what are these figures trying to tell us? Are there any interesting features sticking out here? I would say in general, ***cluster 0*** appears to have quite a few names of people who could be quite important. Immediately, we could start to look at emails from Sally, John, and Eric and see if there is any interesting content. ***Cluster 1*** seems to generally be about meetings with features like chairperson, calendar and time. Again this could be quite useful in terms of narrowing down what emails we want to examine further. ***Cluster 2*** seems to have a lot of words suggesting the emails were from people requesting things. Although ostensibly this doesn't look immediately interesting it could also be worth further investigation. Below is the full Python code for the Kmeans class.

```python

    Parameters
    -----------
        k: int , number of clusters

        seed: int, will be randomly set if None

        max_iter: int, number of iterations to run algorithm, default: 200

    Attributes
    -----------
       centroids: array, k, number_features

       cluster_labels: label for each data point

    """

    def __init__(self, k, seed = None, max_iter = 200):
        self.k = k
        self.seed = seed
        if self.seed is not None:
            np.random.seed(self.seed)
        self.max_iter = max_iter



    def initialise_centroids(self, data):
        """Randomly Initialise Centroids

        Parameters
        ----------
        data: array or matrix, number_rows, number_features

        Returns
        --------
        centroids: array of k centroids chosen as random data points
        """

        initial_centroids = np.random.permutation(data.shape[0])[:self.k]
        self.centroids = data[initial_centroids]

        return self.centroids


    def assign_clusters(self, data):
        """Compute distance of data from clusters and assign data point
           to closest cluster.

        Parameters
        ----------
        data: array or matrix, number_rows, number_features

        """

        if data.ndim == 1:
            data = data.reshape(-1, 1)

        dist_to_centroid =  pairwise_distances(data, self.centroids, metric = 'euclidean')
        self.cluster_labels = np.argmin(dist_to_centroid, axis = 1)

        return  self.cluster_labels


    def update_centroids(self, data):
        """Computes average of all data points in cluster and
           assigns new centroids as average of data points

        Parameters
        -----------
        data: array or matrix, number_rows, number_features

        Returns
        -----------
        centroids: array, k, number_features
        """

        self.centroids = np.array([data[self.cluster_labels == i].mean(axis = 0) for i in range(self.k)])

        return self.centroids



    def predict(self, data):
        """Predict which cluster data point belongs to

        Parameters
        ----------
        data: array or matrix, number_rows, number_features

        Returns
        --------
        cluster_labels: index which minmises the distance of data to each
        cluster
        """

        return self.assign_clusters(data)

    def fit_kmeans(self, data):
        """
        This function contains the main loop to fit the algorithm
        Implements initialise centroids and update_centroids
        according to max_iter
```

Open in app Get started

```
        """
        self.centroids = self.initialise_centroids(data)

        # Main kmeans loop
        for iter in range(self.max_iter):

            self.cluster_labels = self.assign_clusters(data)
            self.centroids = self.update_centroids(data)
            if iter % 100 == 0:
                print("Running Model Iteration %d " %iter)
        print("Model finished running")
        return self
```

Kmeans.py hosted with ❤ by **GitHub** view raw

Some things to keep in mind

It should be clear now that k-means is a simplistic yet powerful algorithm and it can be really useful for many different types of problems that may arise in analytics. With that said, it may not always be the best choice for your particular problem and there are some assumptions that the algorithm makes which you need to be aware of if you are going to use it. Probably the biggest assumption and limitation of k-means is that it assumes that the ***clusters are spherical.*** This assumption essentially translates to all variables having the same variance or in other words, a diagonal covariance matrix with constant variance on the diagonal. If this is not the case, which in practice it often isn't then k-means may not be the best solution. Another limitation with the k-means algorithm is that the data points are ***"hard assigned"*** to a cluster. In other words, the data point is either in the cluster or it isn't. Surely we are more confident about certain data points being in a cluster over others? Wouldn't it be better if we could somehow incorporate this confidence into our results?

Well luckily for us there is another technique we can use to address these issues. We could use an algorithm called ***Gaussian Mixture Modelling or GMM***. The advantage of this is that we end up with soft assignments, i.e. each data point belongs to each cluster with a certain probability. As well as this GMM makes slightly less restrictive assumptions about the variance of the clusters. The downside is it is a more complicated algorithm but this is something I want to discuss further in another post.

Ok, so that's it guys thanks for reading. Hopefully, that has given you all a good understanding of K means as well as how to implement it fully in Python. There are a few extra features we could have implemented in our code such as a ***smart initialisation (k-means++)*** or a better **convergence calculation** for the algorithm but I haven't done these here. How about you guys give it a go? If you want to get an idea of good coding practice I think a great place to start would be the ***GitHub for sklearn***. There are a huge number of algorithms implemented and they have all been tried and tested by the data science community so I encourage people to have a look and maybe try and implement some of them yourself. It is a great way to learn.

Link to the Kaggle Kernel: https://www.kaggle.com/dfoly1/k-means-clustering-from-scratch

Source: *Christopher M. Bishop 2006, Pattern Recognition and Machine Learning*

Source: Bayesian Methods for Machine Learning

Note: some of the links in this post are affiliate links

Sign up for The Variable

By Towards Data Science

Open in app Get started