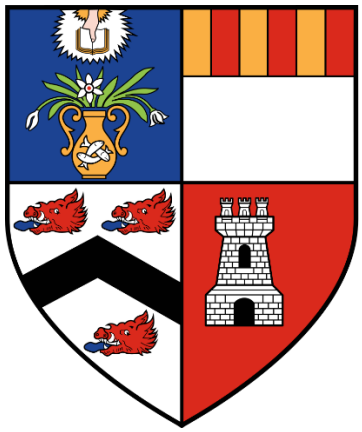


University of Aberdeen

CS551G Data Mining and Visualisation

Dr. Georgios Leontidis

Assessment 1

Submission by
Philip Tsvetanov

ID: 5196769

Task 1 - Clustering

Part 1: Theory

K-means clustering is a type of unsupervised learning which groups data according to their features. K-means is a type of polythetic clustering, which means the cluster members are similar (in more than one feature) and it also uses hard boundaries, meaning that the clusters do not have any overlap. It is a flat clustering technique – there are no hierarchies (Leontidis, 2020).

K-means groups n amount of objects into k clusters, with each cluster's members being the closest ones into its centroid (Sayad, no date b) – this results in a minimum aggregate cluster distance (Leontidis, 2020). This is represented in K-mean's objective function, pictured in Figure 1 (Sayad, no date a). A distance function is calculated for each member – the distance from the member to the centroid – and then all of the distances for all members are summed up and this is then done for each cluster.

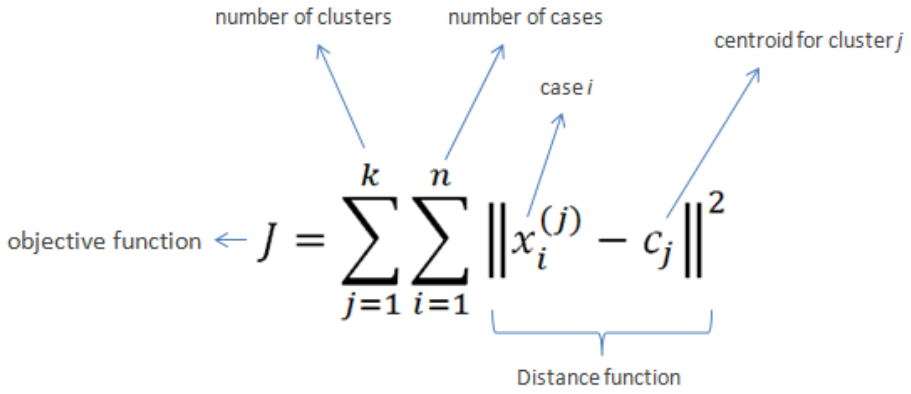


Figure 1 K-means Objective Function

The centroids are similar to the initiation function in a neural net: we can not make an educated guess which positions of the centroids will produce best results, so we attempt several different ones (a random one every new iteration of the algorithm) and measure the outcomes.

To do this we need to use a distance metric and there is a few of them that produce different results. Similarly to kNN, Euclidian distance is one of the most popular distance metrics. Once the Euclidian distance is measured from each centroid to the data point, it is assigned to the closest centroid.

When all the points are assigned to a cluster, the mean of all points in a cluster is set as its new centroid. And the distance and assignment procedure is repeated until convergence is reached – the updated centroids of the clusters are the same as the current centroids.

The Gaussian Mixtures, on the other hand, has soft boundaries, but still clusters members in a polythetic manner and is a flat clustering algorithm. It works by using mixtures of gaussian values, fits them to the data and uses probability to define the cluster's boundaries. Each member then has different membership values to each cluster, and can be considered part of the cluster it has the higher membership value to. This is seen as a more flexible approach.

Part 2: Data Exploration

The dataset was uploaded to Google Drive and accessed through there from Google Colab using the appropriate commands.

```
dataframe = pd.read_csv("DMV_CS551G_Assessment1_Task1_dataset.csv")
#dataframe.head(10)
```

A quick histogram plot is made for each of the features of the data set.

```
dataframe.hist(bins=50, figsize=(20,15))
plt.show
```

As seen from Figure 2 below, the "height (top left) and tail length (bottom right) features follow a more or less standard distribution (bell-curved) shape. The height feature is more spread out towards the two ends, but the mean is slightly skewed towards the smaller numbers – there are more records in the 5.0 – 6.0 range, that there are in the 6.5-7.5 range. The tail length is also skewed towards the smaller numbers (the most records around the 2.7-3.2 range), but it is important to note that there are quite a few more records on the opposite (right hand) side, which means that both the average and the mean would be higher.

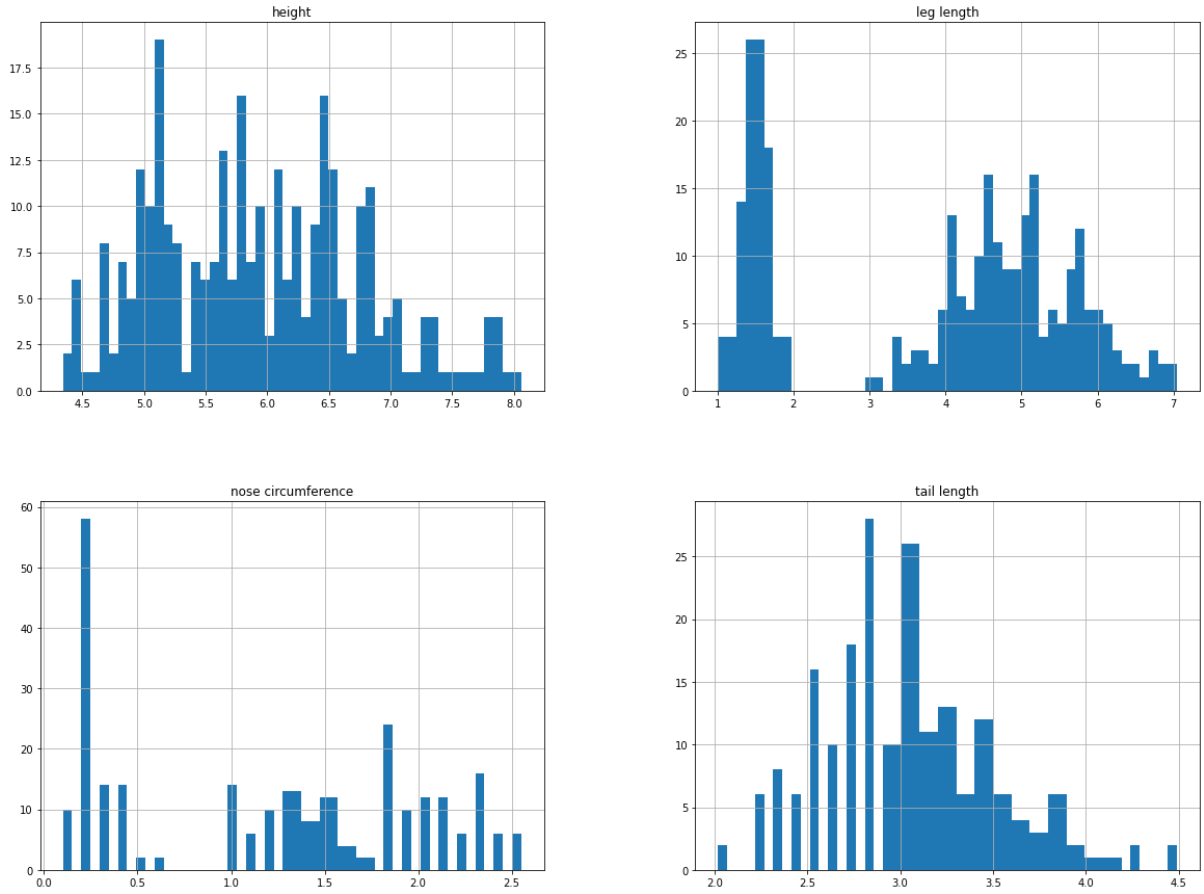


Figure 2 Histograms of each feature of the dataset

The leg length and nose circumference features are very different and interesting. The former shows a clear division of the data into two "groups" – a narrower (range of 1.0-2.0) group and a wider, but less densely populated group at the ranges 3.0-7.0. This could imply that the leg length is correlated with some other factor, e.g. male dogs being taller on average, but with quite some variance, compared to female dogs that are shorter, but are more consistent with their leg length. The nose circumference on the other hand is arguably split into 3 groups, with small (0-0.5), medium (1.0-1.7) and large (1.7-2.5) noses, although it could be argued that the last two are two parts of a bigger group. There is a disproportionate amount of representatives in the 0.2 range, compared to the rest of the data.

The describe() command (below) was used to reveal some statistical information about the dataset.

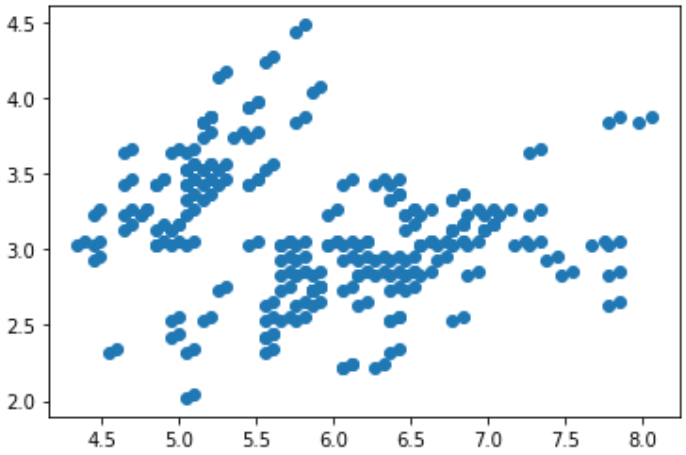
```
dataframe.describe()
```

The results can be seen in the table below. There are 300 entries with the four different features. One noteworthy fact is the size of the standard deviation for the leg length: more than double of the second highest one – height. A reflection of this can be seen in the quartiles: the 25% quartile falls at only 1.6, where as the 50% quartile is all the way at 4.4. This is undoubtedly an effect from that large group of data points around the 1.0 range we saw earlier.

	height	**tail length**	**leg length**	**nose circumference**
count	300.000000	300.000000	300.00000	300.000000
mean	5.930983	3.103193	3.81437	1.217323
std	0.839601	0.441935	1.78890	0.772409
min	4.343000	2.020000	1.01000	0.101000
25%	5.202000	2.828000	1.61600	0.305250
50%	5.858000	3.060000	4.41500	1.326000
75%	6.528000	3.366000	5.20200	1.836000
max	8.058000	4.488000	7.03800	2.550000

We can visualize the data on a scatter plot to see if some clusters of the data can be seen before we continue forward. Scattering the points in different ways usually separated the data in two large and not well-defined clusters. It is only by plotting the 1st and the 2nd features – heights and leg length – that we saw more interesting results. See code below.

```
plt.scatter(data[:,0],data[:,1])
```



The code produced the plot that can be seen on Figure 2. As we can see, there is definitely a well-defined cluster at the top-left part of the graph, and at least one more large cluster at the bottom-right. However, it could be argued that the large cluster is quite segmented and could in fact be comprised of many smaller clusters. To explore this, we will need to employ the help of clustering techniques.

Figure 3 Scatter plot of height vs leg length

Because these two features present the best scatterplot (open to different interpretations when it comes to clusters), it will be used to demonstrate the clustering of the data in the following chapters.

Part 3: K-Means

The model used for K-means was scikit's "sklearn.cluster.KMeans" model. A function that calls this model has been developed, the benefit being that it can be called with different hyperparameters.

```
model = KMeans(n_clusters=K,n_init = 100)
```

In the code above, the aforementioned model is created, where n_clusters is the parameter of the model that determines how many clusters to create and n_init is the the number of times the clusters will be created with different centroids. The model will choose the best position of the centroids to provide the final results. This model is being ran in a function – this allows me to call it multiple times with different parameters and conduct experiments faster. In this case, the number of clusters equals K, the parameter passed to the function from the main scope of the program.

```
model.fit(data)
labels = model.labels_
# sum of squared distances of samples to their closest cluster centrer
distance = model.inertia_
```

Once the model is created, we fit the data to the model and retrieve the labels and the inertia – the sum of the squared distance from all the samples to their closest centroid. This important for measuring the cluster validity, since we have no data labels provided.

```
 cluster_set = {}
    for i,x in enumerate(data):
        xlabel = labels[i]
        if xlabel not in cluster_set.keys():
            cluster_set[xlabel] = np.reshape(x,(1,len(x)))
        else:
            cluster_set[xlabel] = np.append(cluster_set[xlabel],np.resh
ape(x,(1,len(x))), axis=0)
```

```
    return distance, cluster_set
```

Then we create a dictionary for all the clusters and add each of their members the corresponding entry of the dictionary.

```
plt.figure(1)
colors = {0:'k',1:'b',2:'g'}
center_color = ['r']
for key in clusters_set.keys():
    cluster = clusters_set[key]
    plt.scatter(cluster[:,0],cluster[:,1],alpha=0.5)
```

Once we have called the function and received these clusters, we can create a scatter plot, similar to the one above, but with colour-coded clusters. We can repeat this two times and K=3 and K=4.

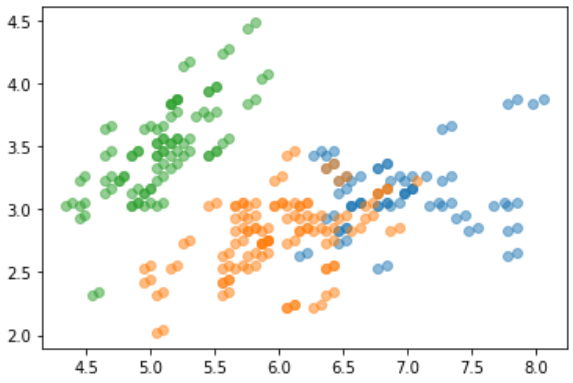


Figure 4 Clusters for K=3

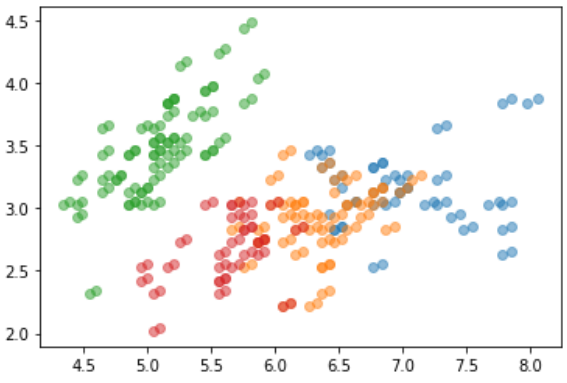


Figure 5 Clusters for K=4

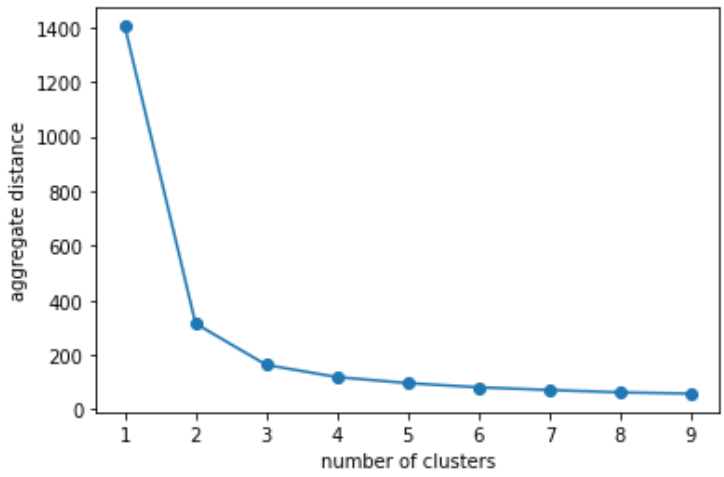
As we can see for Figures 4 and 5, both options provide a plausible separation of the "bigger cluster". The clustering for K=4 is quite cluttered, but other than that it is hard to tell how many breeds the data contains. To give us more insight, we need to calculate the aggregate distance between the points – we have done this in the lab and some of the code for it is being reused.

```
def agg_distance(data):
    all_distance = []
    for k in range(1,10):
        vector = []
        vector.append(k)
        distance, clusters_set = kmeans_clustering(data, k)
        vector.append(distance)
        all_distance.append(np.array(vector))
    return np.array(all_distance)
```

By using the functions from Lab 2, we can also calculate the aggregate distance of the model for K=(1:9). This means Summing the distances between all the points and their cluster centroids.

```
all_distance = agg_distance(data)
plt.figure()
plt.plot(all_distance[:,0],all_distance[:,1],'o-')
plt.xlabel('number of clusters')
plt.ylabel('aggregate distance')
```

We then plot the numbers in a simple line diagram for a better representation.

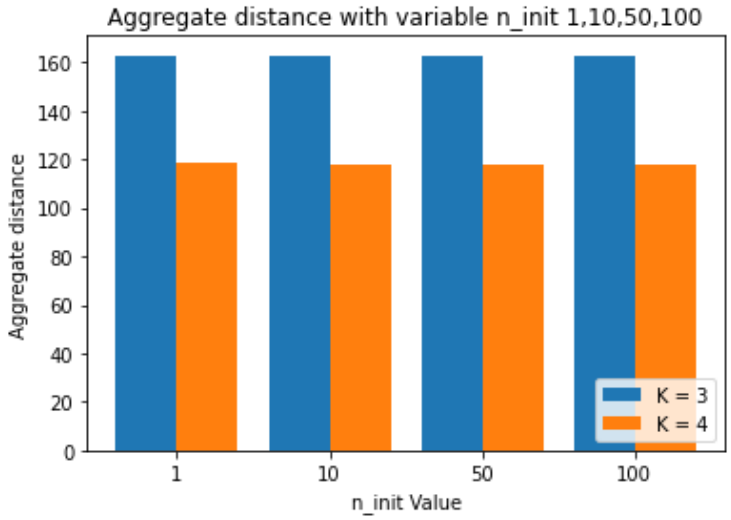


As we can see from Figure 6, there is a massive improvement moving from K=1 to K=2, and then we start to experience diminishing returns: each further separation benefits the aggregate distance only slightly. K=3 is still a good improvement, but K=4 and anything after that provides very little improvement.

Figure 6 Aggregate distance for K=(1:9)

```
#Tuning N-init
paramList = [1,10,50,100]
def kmeans_clustering_n_init(data, K):
  for v in range(len(paramList)):
     model = KMeans(n_clusters=K, n_init=paramList[v])
```

To explore whether an increase in the number of times the algorithm is ran with different centroids would change the outcome, the model is being ran for each value in the parameter list. This includes n_init=1,10,50,100. A bar diagram has been created with the result for K=3 and K=4, for each value.

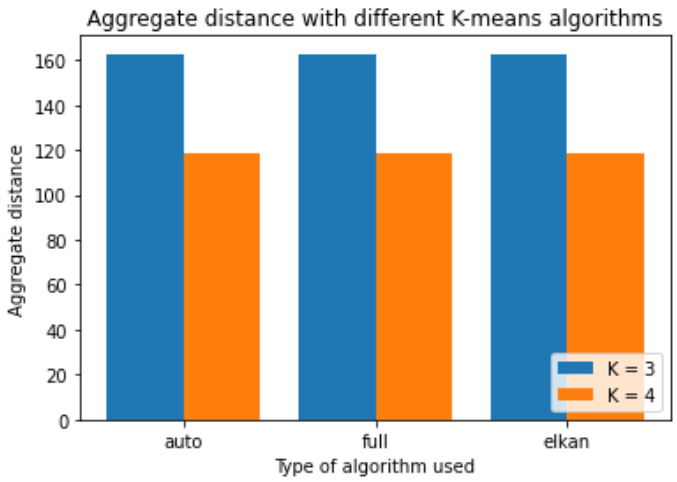


As seen from Figure 7, changing the n_init value did not provide any change to the distance metric, it was 162.8 and 118.2 for K=3 and K=4 respectively. It has also been attempted with an order of magnitude higher numbers (n_init = 100, 200, 500, 1000), but there was no change from the default n_init = 300 either.

Figure 7 Aggregate distance with n_init=1,10,50,100

```
paramList = ['auto', 'full', 'elkan']
def kmeans_clustering_alg(data, K):
  distances = []  for v in range(len(paramList)):
     model = KMeans(n_clusters = K, algorithm = paramList[v], precompu
te_distances = True)
```

The next attempt is changing the K-means algorithm to use. These algorithms vary in their implementation (some are better for bigger or sparser data), but should provide identical results.

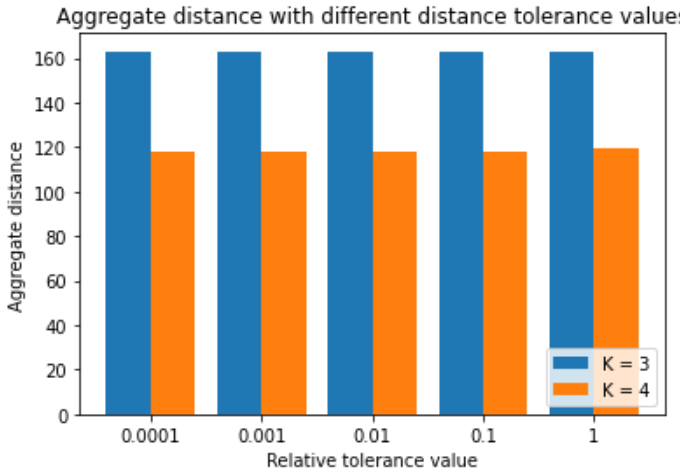


As we can see from Figure 8, the different algorithms provide identical performance.

Figure 8 Aggregate distance with different K-means algorithms

```
paramList = [0.0001, 0.001, 0.01, 0.1, 1]
def kmeans_clustering_tol(data, K):
  for v in range(len(paramList)):
      model = KMeans(n_clusters = K, tol = paramList[v])
```

After it was attempted to tune the relative tolerance to declare convergence in regards to the inertia. This test was undertaken with the values 0.0001, 0.001, 0.01, 0.1, and 1, where the default for this parameter is 0.0001.



As seen from Figure 9, no difference in performance was observed here either.

Figure 9 Aggregate distance with different convergence tolerance values

Part 4: EM Clustering

The gaussian mixture was implemented in a similar way, with the exceptions that it does not have the inertia attribute, so the distance needs to be calculated manually and it does not have centroids like K-means, so mean of the clusters are being used.

```
def calculateDistance(means, clusters):
  distance = 0
  for x in clusters.keys():
    for point in clusters[x]:
      distance += np.sqrt(sum(np.square(point - means[x])))
  return distance
```

This is the function that calculates the aggregate distance for each cluster. Notice that this implements the sum of squared distances metric, similar to Euclidean distance – the coordinates of each member, minus the coordinates of the means, squared (to negate any negative distances), summed up and then square rooted.

```
distances.append(calculateDistance(model.means_, clusters_set_em))
```

The means are an attribute, accessible from the model, similar to the centroids of K-means. The cluster set is gathered by a for-loop the same way as it was in the K-means solution above.

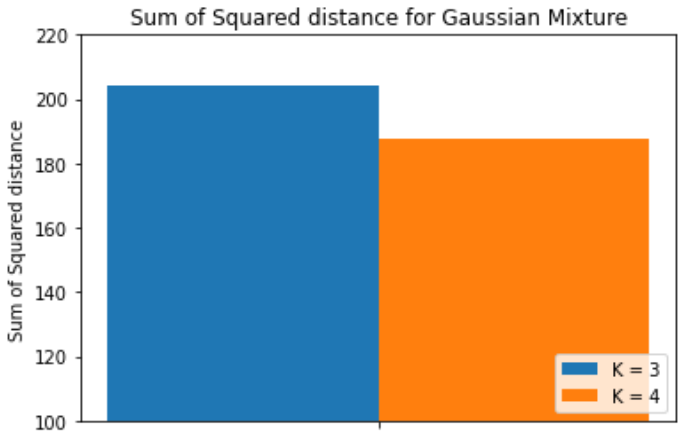


Figure 10 Sum of Squared distance for Gaussian Mixture

The results are presented in Figure 10. It seems as though the GM algorithm is performing quite worse compared to K-means with values of 204.3 and 187.67 for K=3 and K=4 respectively. Note that the scale of the graph is ylim=(100,220), so the difference would realistically be smaller – it is just made this way to make it easier to observe.

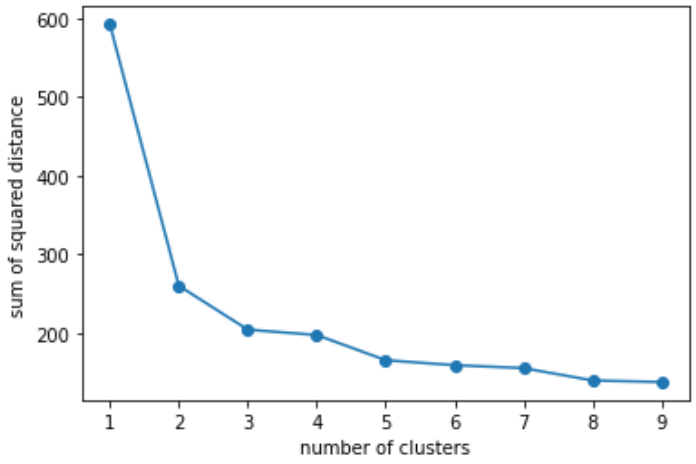


Figure 11 Sum of squared distance for EM, K=(1:9)

On Figure 11 we can see the Summed square distance for the EM algorithm, similarly to what was shown for K-means above. Similar results can be observed here – not by absolute values, but by ratio. The "diminishing returns" effect is in full force, and there would be no reason to chose K=4 over K=3 here. The plot is made the same way as the previous one, but the new em_clustering function has been used.

Next we are going to have a look at fine tuning the EM algorithm.

```
paramList = ['full', 'tied', 'diag', 'spherical']
```

```
for param in paramList:
  model, clusters_set_em = em_clustering(data, 3, param)
```

Firstly, the covariance type – whether or not the components have their own covariance type (full/diagonal), the all share the same covariance (tied) or each component has its single covariance (spherical).

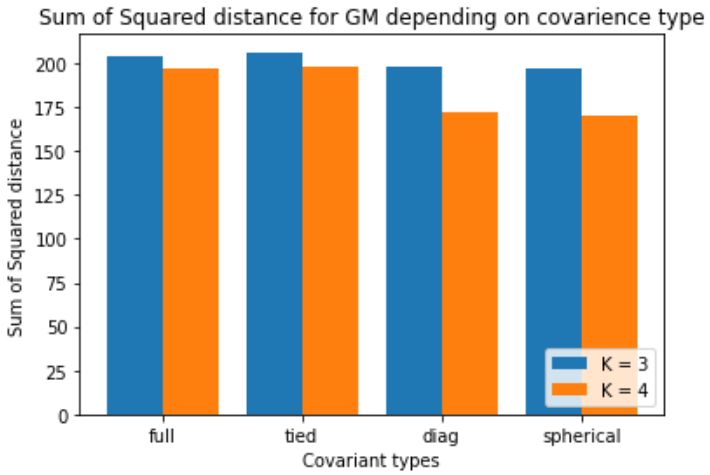


Figure 12 Sum of Squared distance for GM depending on covariance type

Figure 12 shows the results: slight variances in performance, with tied being the worst by a small margin and spherical being just slightly better than diagonal (197.42 and 170.6 for K=3 and K=4 respectively), increasing the performance slightly, from the default "full" covariance (204.3, 183.64). Because of this, the spherical type has been used for the rest of the experiments below.

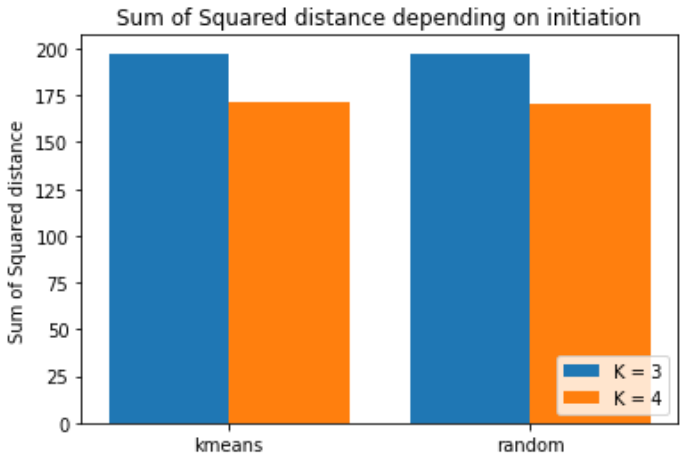


Figure 13 Sum of Squared distance depending on initiation

The next experiment would be with the different ways to initialize the parameters of the algorithm (the weights, the means and the precisions). The default way of doing this is through the standard K-means methodology, but it can also be done randomly. The results are shown on Figure 13. The random option would produce consistent result, unlike the default – and that makes sense, given that it is "random".

No random seed was used in this experiment, since this would just produce a constant (but very one-dimensional) result for the random initiation, instead the tests have been ran a few times, letting the instance's random number generator have its way. No specific results were recorded (other than the ones on the diagram), because variance was very rare: the two algorithms would usually perform equally, with the random initialiser sometimes performing a bit worse for K=3 or slightly better for K=3.

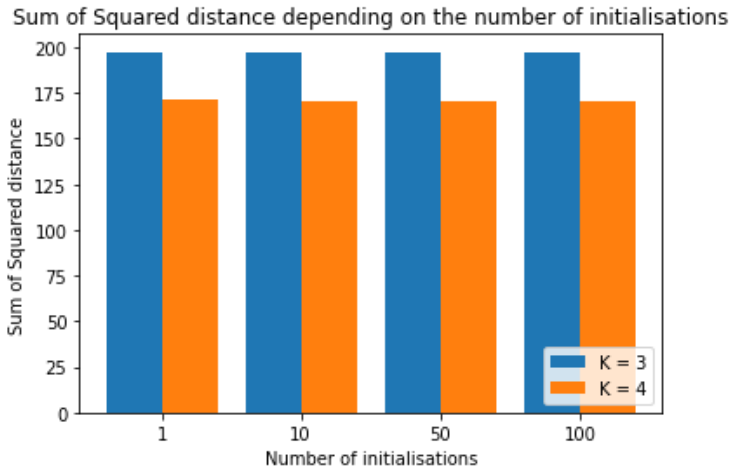


Figure 14 represents an experiment with different number of initialisations (n_init), similar to the ones done for K-means above. This test went the same way as before: no performance difference was noted with the numbers (1,10,50,100), nor with these numbers with a order of magnitude larger (10,100,500,1000).

Figure 14 Sum of Squared distance depending on the number of initialisations

Part 5: Conclusion

In general, the findings from fine-tuning the parameters were not very exciting. In general, clustering would be done for larger datasets, and although it works here, there is no big difference (or any at all) registered for things like n_init, which would probably be very significant for harder and bigger datasets. To illustrate this, even n_init=1 worked sufficiently well for this example, something that should, in theory, have drastically poorer performance for bigger datasets. It seems like clustering of this one does not require specific placement of the centeroids/means, and the algorithm is making a very good first guess.

The sum of squared distances for the EM algorithm is quite poor compared to the aggregate distances of the K-means, which is surprising. However, since I had to implement that myself, the possibility of a coding mistake is persistent. Despite this, result are generally very consistent and rarely change with parameter tuning – similar to K-means.

In conclusion, looking at the distance per K graphs and the spread of clusters (colour-coded) diagrams, it is hard to decide whether there are 3 or 4 dog breeds in the data set. It would also depend greatly on which dog breeds they are, since breeds are not always binary as some might think: some breeds area crossbreeds between existing breeds, and therefore have a lot in common with them. For example, there are numerous types of Husky, depending on its parents, and Huskies themselves are a close relative to the polar wolf. This makes it a really hard decision, but in general, the graphs show very small variance and lead me to believe that only three breeds are included.

Task 2 – Time Sequences

Part 1: Data Exploration

Firstly, the files provided are imported through the colab files tool:

```
from google.colab import files
files.upload()
```

There are 216 unique csv files. Because it is hard to work with different files, all the time sequences from the files are extracted into a large list:

```
path = '/content/Ambient_Living_'
sequences = list()
```

```
for i in range(1,217):
    file_path = path + str(i) + '.csv'
    print(file_path)
    df = pd.read_csv(file_path, header=0)
    values = df.values
    sequences.append(values)
```

Exploring the data, it has only 4 fields, all of them comprised of sensor data, that contains positive and negative decimals. One of the files also contains the labels for the data – 216 instances of 1 and -1 (perhaps representing movement or lack of movement).

	Sensor 1	Sensor 2	Sensor 3	Sensor 4
0	0.28571	0.76	-1.00000	-0.55
1	0.47619	0.64	-1.00000	-0.55
2	1.00000	0.60	-0.80952	-0.55
3	0.28571	0.16	-0.76190	-0.60
4	0.14286	0.80	-0.76190	-0.60

The labels are extracted from the file and saved as a separate list.

```
targets = pd.read_csv('/content/Ambient_Living_target.csv')
targets = targets.values[:,1]
```

This is quite complicated data, but its use will be made easier if it is properly structured. Each unique entry will be treated as a vector of numbers and all of these vectors within a csv would be treated as sequences (of vectors). This will preserve their continuity and the knowledge that comes from it, however the sequences are in no particular order: even if they have been recorded sequentially, that continuity is less important, or perhaps irrelevant. This will be explored in the following experiments.

```
len_sequences = []
for one_seq in sequences:
    len_sequences.append(len(one_seq))
pd.Series(len_sequences).describe()
```

The sequences need to be padded and cropped so to explore their length, a big list of all the sequences is created through the use of a loop. The describe function provides the following information:

```
count    216.000000
mean      37.407407
std       14.948306
min       19.000000
25%       25.000000
50%       33.500000
75%       47.750000
max      103.000000
dtype: float64
```

The Count, min and max rows are self-explanatory. “std” is standard deviation, which measures how much they deviate from the average. The 25%, 50% and 75% rows show the corresponding quartiles. For example, 25% of the sequences are shorter than 25, while 50% are lower than 33.

The maximum length is 103, this will be important for the padding.

```
#Padding the sequence with the values in last row to max length
to_pad = 103
new_seq = []
for one_seq in sequences:
    len_one_seq = len(one_seq)
    last_val = one_seq[-1]
    n = to_pad - len_one_seq
    to_concat = np.repeat(one_seq[-1], n).reshape(4, n).transpose()
    new_one_seq = np.concatenate([one_seq, to_concat])
    new_seq.append(new_one_seq)
final_seq = np.stack(new_seq)
```

Here, similarly how it was done in the exercise of Lab 5, the sequences are padded with empty data up to length of 103 – the length of the longest sequence.

Now they need to be truncated to the 80th percentile. To find the 80th percentile, the .quantile() function is used:

```
pd.Series(len_sequences).quantile([.80])
```

This returns a value of 54 – the sequences must be truncated at 54 values.

```
#truncate the sequence to length 54
from keras.preprocessing import sequence
seq_len = 54
final_seq=sequence.pad_sequences(final_seq, maxlen=seq_len, padding='po
st', dtype='float', truncating='post')
```

This is done through the keras.preprocessing.sequence package.

To develop the initial model, the data is simply split into a 80/20 ratio:

```
train = [final_seq[i] for i in range(1,172)]
test = [final_seq[i] for i in range(172,216)]

train_target = [targets[i] for i in range(1,172)]
test_target = [targets[i] for i in range(172,216)]
```

Part 2: Baseline Model

For the baseline model, a simple LSTM model is created with a LSTM layer with 256 units and a single dense layer with 1 as value.

```
model = Sequential()
model.add(LSTM(256, input_shape=(seq_len, 4)))
model.add(Dense(1, activation='sigmoid'))
```

The Adam optimiser is used, with a learning rate of 0.001, and a checkpoint to save the best model is created. The model history can also be saved in a variable for further inspection.

```
adam = Adam(lr=0.001)
chk = ModelCheckpoint('best_model.pkl', monitor='val_loss', save_best_o
nly=True, mode='min', verbose=1)
```

```
model.compile(loss='binary_crossentropy', optimizer=adam, metrics=['acc
uracy'])
h = model.fit(train, train_target, epochs=200, batch_size=128, callback
s=[chk], validation_data=(test,test_target))
```

The baseline will be using 200 epochs and batch size of 128. The loss will be compiled using binary entropy and accuracy will be the main metric that guides the model.

This baseline model performs very sporadically, with accuracies from 80 to 93 percent (recorded just from a few runs) and this hints that cross-validation should be used for the task. Nevertheless, we can take the model's history and plot the loss of each epoch:

```
loss = h.history['loss']

plt.figure(figsize=(20,5)) #width, height settings for figures
plt.plot(range(1, len(loss) + 1), loss, marker='o')
plt.xlabel('Epochs')
plt.ylabel(Loss')
plt.show()
```

The result can be seen in Figure 15. The graph shows graphs of how loss progresses over the epochs, as the model trains. As it can be seen, these vary massively from run to run, but there is a trend: the model tends to fit just before 100 epochs, so perhaps we can lower the epochs to save cocmputational time.

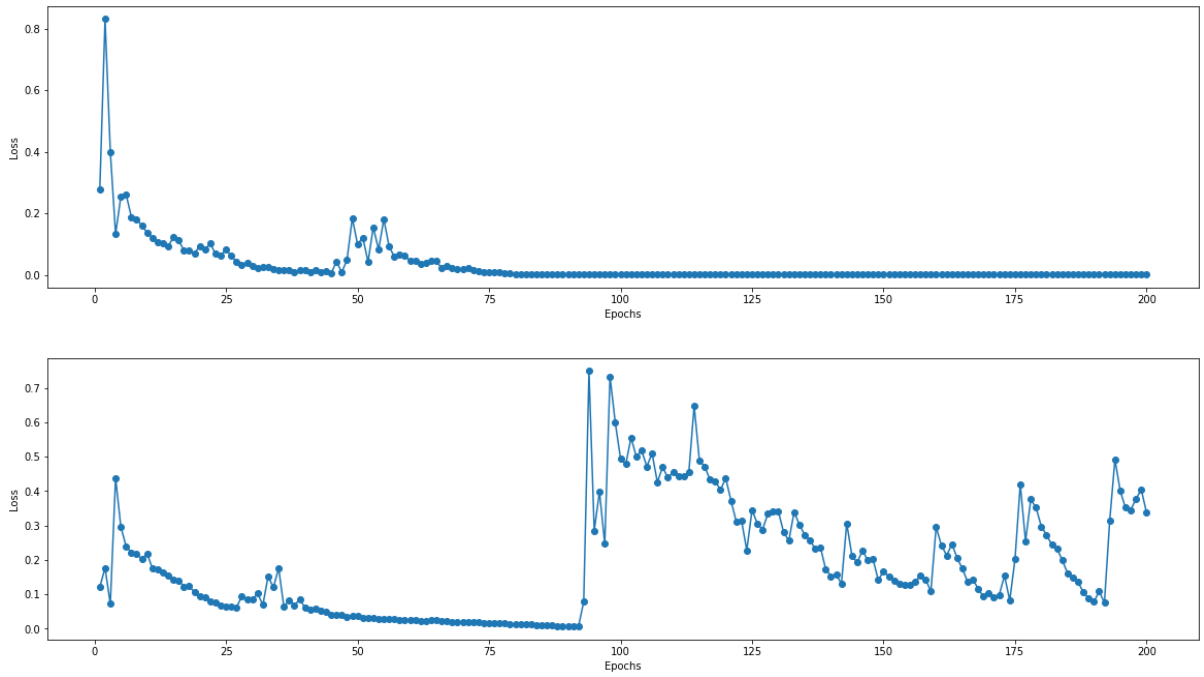


Figure 15 Loss vs Epochs

The next step is to implement cross-validation to lower the run-to run variance, by averaging the result the different folds.

```
kfold = StratifiedKFold(n_splits=10)
    score = 0
    for train_idx, test_idx in kfold.split(final_seq, targets):
      train_target = np.array(targets[train_idx])
```

```
        test_target = np.array(targets[test_idx])
```

StratifiedKFold is used, a cross-validation method that samples equal amounts of data (based on their label) for the training on the testing. The training and testing indexes that are used to sample the training and testing data are provided by the kfold.split() function. 10-fold cross-validation is used, which standard and more than enough for such a small dataset. To calculate the average accuracy, the score from each fold is summed up and at the end divided by 10 (the number of folds):

```
scores.append(score/10)
```

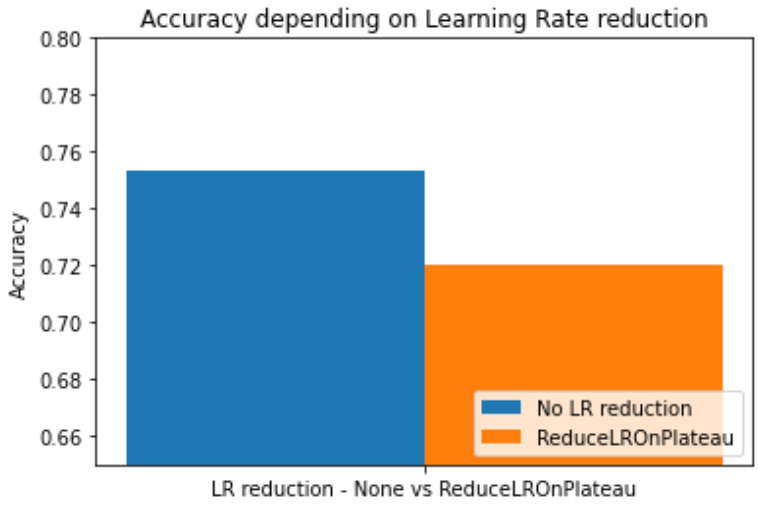
The method developed so far would be the baseline for the next part, tuning the parameters of the model and exploring different models.

Part 3: Parameter fine-tuning

Firstly, an experiment with an additional callback was undertaken, similar to how it is done in this Kaggle blog post. The ReduceLROnPlateau callback does exactly what the name suggests: it reduces the learning rate if the model has stopped learning – reached convergence – to inhibit it from overfitting.

```
reduce_lr_loss = ReduceLROnPlateau(monitor='loss', factor=0.1, patience
=10, verbose=1, epsilon=1e-4, mode='min')
```

The callback here is monitoring the loss, reducing it by 0.1 if no improvement is seen in 10 epochs (the "patience" parameter). The epsilon is the difference that the new model needs to be better, in order for the LR to not be reduced.



As seen from Figure 16, for this scenario the LR reduction callback lowers the accuracy of the algorithm. This means that no LR reduction is needed, since reducing the learning rate only makes the model less accurate in this instance.

Figure 16 Accuracy depending on LR reduction callback

The next experiment has to do with shuffle of data. As mentioned previously shuffling the files should not reduce the accuracy, unless there is some long sequential dependency between each sequence, but even then it is unlikely that the LSTM can make use of that, since its memory would learn from the vectors themselves. Shuffling, on the other hand, hand can be beneficial, because the data is ordered in clusters – a lot of True and False values are stuck together – increasing the chance to influence the model.

```
seed = 420
np.random.seed(seed)
# define 10-fold cross validation test harness
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
```

The stratified k-fold has a "shuffle" parameter, which must be set to true, but when doing this, a random seed should be set. This is important for tasks that require randomness, because it makes the results easy to reproduce. The downside, of course, is that this is not very representative: if this seed yields bad results for one of the models due to variance, then that model would be at a disadvantage against other ones.

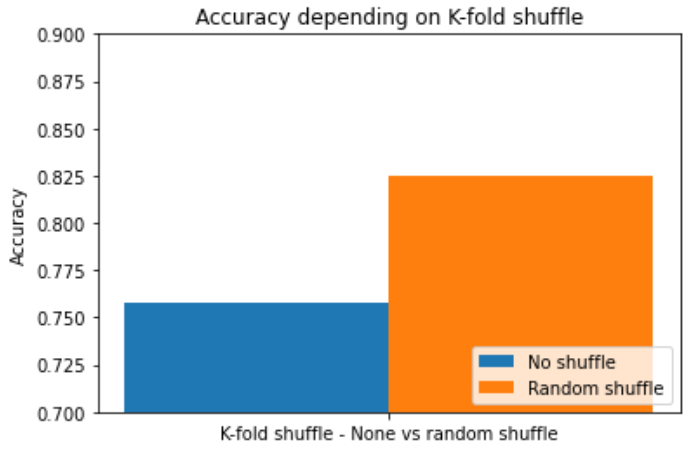


Figure 17 Accuracy depending on random shuffle

Illustrated on Figure 17 are the two models – with and without shuffle. It is obvious that the model using the shuffle performs much better (7% on average) and that the above hypothesis is correct. For the following experiments, the random shuffle will be used as part of the baseline model.

The next experiment will be looking at the baseline model with different epochs. As mentioned previously, the models seems to reach convergence at 100 epochs, so running it any longer might not make sense, especially if it starts overfitting. Figure 18 shows the results with a test carried out with 50,100 and the baseline 200 epochs. There is a small difference between the first two, which is not too surprising if the above graphs are observed again – 50 epochs brings the loss down close to the levels of 100, but 200 epochs brings the average accuracy down. For the next experiments 100 epochs will be used a the baseline.

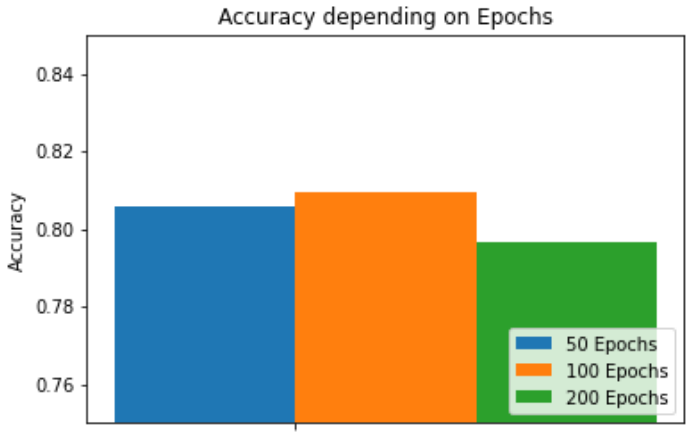


Figure 18 Accuracy depending on epochs

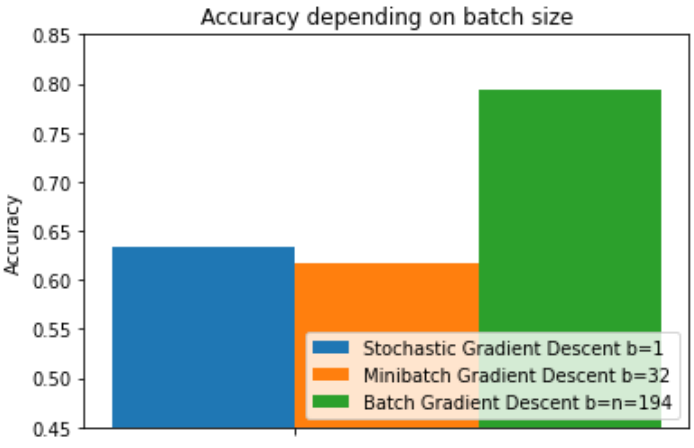


Figure 19 Accuracy depending on batch size

On Figure 19 we see the results of experimenting with different batch sizes. According to my research, there is three types of batching – Stochastic Gradient Descent – when the model is trained 1 batch at a time, Minibatch Gradient Descent – when the n > batch_size > 1, where n is the size of the training data and Batch Gradient Descent, where batch_size = n. Here we see that the Batch gradient descent performs the best and the other two models fall in accuracy quite

noticeably. Minibatch of size 32 has been selected for the model, because according to the same website that works best for many models and algorithms and I have noticed a few examples where it has been used, even in our lecture exercises.

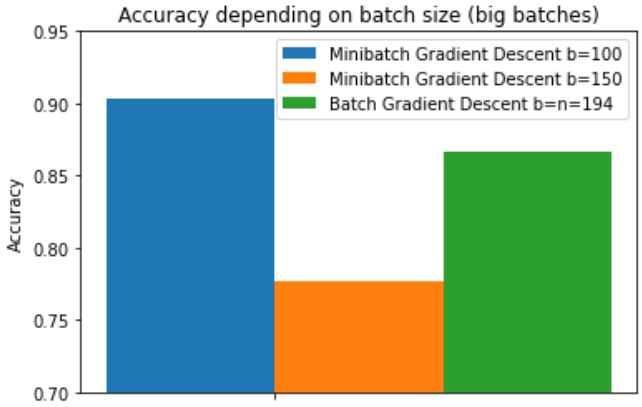


Figure 20 Accuracy depending on batch_size (big batches)

With the minibatch result above being surprisingly low, another batch-size test was conducted, this time with two bigger minibatches (batch_size = 100,150) and the batch gradient descent for reference. It seems that having the batch size roughly half of the training data (b=1/2*n) produced the best result – 90% accuracy (although it might vary) and this will be used in further experiments.

In general, models using the default learning rate of algorithms perform the best, and there is rarely any improvement in tuning it. It is still an interesting experiment to perform, especially with a small dataset, where the model learns very quickly. To make the experiment more exciting the LR reduction from earlier is also added as a factor, with the different learning rates tested with both LR reduction and without it. For the learning rates themselves, an order magnitude higher and lower than the default were chosen, to cover both ends of the spectrum.

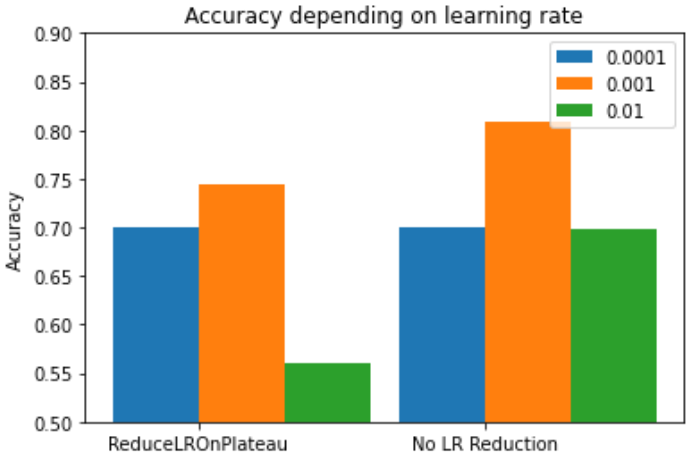


Figure 21 Accuracy depending on LR, with and without LR reduction

The results are shown on Figure 21 and it seems that the default learning rate of 1e-4 unsurprisingly performs the best. This might be due to the fact that the model has been tuned to 100 epochs, where it performed best. If more (or less) epochs were used, perhaps some of the other options would have been better (more epochs, higher LR or vice versa).

The final experiment for the parameter tuning will take a look at the different optimisers that can be found in the Keras documentation. They are quite numerous, so I will not go through them one by one, but rather explore any that perform better than the default, “Adam” optimiser.

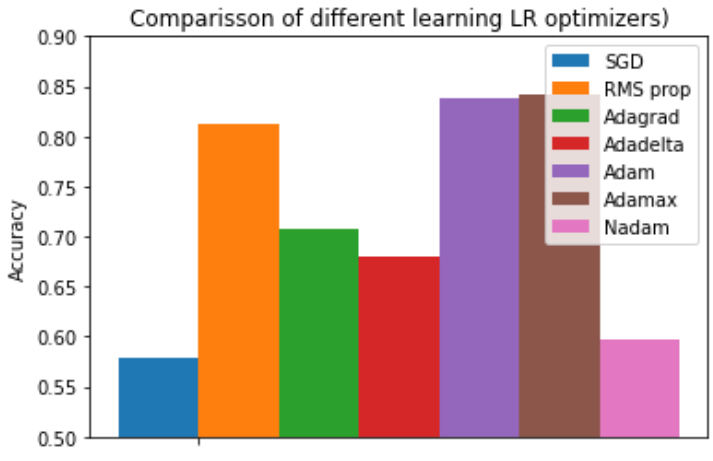


Figure 22 Accuracy depending on different optimisers

Unsurprisingly, the Adam optimiser provides one of the highest accuracies from the group. RMS prop, which divides the gradient by an average of its recent magnitude, provides a good result. The only one that fares better than Adam, however, is Adamax, which is a variant of Adam based on the infinity norm. That being said, the difference between the two is very small and can probably be negated by variance.

Part 4: Deep LSTM

A deep LSTM model was created using two LSTM layers. Firstly, a input layer had to be created with the shape of the input data:

```
model.add(InputLayer(input_shape=(seq_len, 4)))
```

After that, the second layer, the first LSTM was added, with the parameter return_sequences=True, to allow it to transfer the whole input sequence, rather than the last abstract hidden layer:

```
model.add(LSTM(256, return_sequences=True))
```

This allows adding further LSTM layers after this one:

```
model.add(LSTM(128))
```

The next one has half of the units and return_sequence set to False (the default), since this is going to be the last LSTM layer. It is a general practice to add half of the units to each consecutive layer, to refine the weights of the algorithm.

Finally, the last layer is added – the Dense layer, similar to previously:

```
model.add(Dense(1, activation='sigmoid'))
```

The experiment was ran with two LSTM layers, once with 512 and 256 units and once with 256 and 128 units and the average accuracy of those models is 87.5% and 84.6% respectively.

Part 5: Conclusion

The data was explored and preprocessed, according to instruction. A baseline LSTM algorithm was developed for the classification and cross-validation was added to it. This baseline had an average accuracy of about 75%. It was found that LR reduction callback has a negative effect both on the default learning rate of 1e-4 and on bigger or smaller values. As hypothesised, shuffling the data at the k-fold step provided higher accuracy of around 82% on average. An observation was made that the model converges in about 100 epochs and started to overfit in 200 epochs. A change in the epochs from the default 200 to 100 did not provide additional accuracy, but it shortened the computation time by half.

Finding the optimal batch_size – Minibatch Gradient Descent with size 100 for each minibatch increased the accuracy to 90%. Changes in learning rate and optimisers had no effect on the results.

There also was an attempt to create a deep LSTM model with two layers, however it did not produce better results. This might be because the dataset in hand is small and maximum performance is gained just from a single layer. It could be the case that the parameter-tuning in the third chapter tuned this model to produce the best results for a single layer, and if these deep models are fine-tuned in a similar manner, even higher (>90%) accuracy can be achieved.

References

Leontidis, G. (2020) 'Lecture 2 – Clustering'.

Sayad, S. (no date a) 'Clustering K-means diagram'. Available at: https://www.saedsayad.com/clustering_kmeans.htm.

Sayad, S. (no date b) *K-Means Clustering*. Available at: https://www.saedsayad.com/clustering_kmeans.htm (Accessed: 14 April 2020).