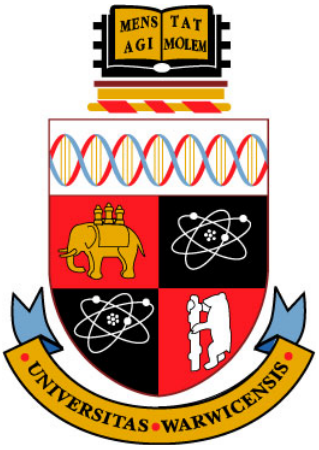


deltaDEC: A Linear Approach to Deep Embedded Clustering (DEC) Based on the Delta Distribution

3rd Year Essay

Thomas Bidewell

Supervisors:

Prof. Susana Gomes (University of Warwick)

Olivier Go (ParlayPlay)

Contents

Cluster analysis is a cornerstone of machine learning, involving the grouping of unlabelled data and spanning a diverse range of applications from image detection to customer segmentation. The demand for high-quality clusters has led to the integration of deep learning techniques into the clustering algorithm, exemplified by Deep Embedded Clustering (DEC) in 2016 [1]. Despite their success, these methods require immense computational resources making them largely impractical for industrial use. This essay demonstrates that scaling down these techniques does not retain their ability to perform high quality clustering and that simpler methods may be more appropriate when working on a smaller scale. Specifically, we propose deltaDEC, a small, quick to train linear adaptation of DEC that uses the Nearest Cluster Delta Distribution. We present its ability to produce near perfect clustering on real world, industrial scale data for a fraction of the computational cost. Our research could be of significant importance for companies looking to incorporate state-of-the-art clustering techniques into their data analytics without the associated computational cost.

1 Introduction

Understanding one's customer base is paramount for companies, providing valuable insights into areas where improvements can be made. In collaboration with ParlayPlay, a Daily Fantasy Sports company founded in 2021 by Olivier Go, Florens Roell, and William Pullen, this essay explores the application of modern clustering techniques to an industrial setting through an analysis of ParlayPlay's user base. With its rapid growth, boasting 30,000 to 40,000 new users monthly, ParlayPlay possesses a substantial amount of untapped data — approximately 315,000 users' worth - holding the potential for revealing interesting results.

At its heart, *Machine Learning* (ML) concerns a mathematical model designed to perform a task, which is then evaluated and improved based on how well it performs this task on specific data - a process known as *learning* [2]. If this data contains the true outputs of the task, this is *Supervised Learning* [3], a sub-genre of ML with wide reaching applications. Image detection, for example, falls under this category and has been harnessed for the likes of cancer detection [4] and self-driving cars [5]. When the true values are not present in the data, we have *Unsupervised Learning* [6], which is the case for this essay. The most prominent method in this category is *clustering*, where the model aims to group similar data-points without prior knowledge of labels for the data. There are many common, traditional approaches to this task, such as KMeans [7], but recent advancements have been made using *deep learning* [1, 8], known as *deep clustering* [9]. This form of learning uses *Neural Networks* [2] (discussed at length in MA3K1) and is key to many major Artificial Intelligence (AI) breakthroughs of the last decade, for example Large Language Models [10].

This essay begins by discussing ParlayPlay's data before covering the core ML techniques used in clustering and methods to improve upon these traditional approaches. Each new model is evaluated by its performance on our data using cluster-specific metrics defined in the essay. We then discuss the mathematical theory behind the deep learning algorithms and explain the current state-of-the-art (SOTA) approach for deep clustering. Following this, we propose *deltaDEC*, a light-weight and effective modification to the current SOTA approach, showcasing its potential at producing near perfect clusters for a fraction of the computational cost. We then analyse deltaDEC's clustering in a business context and conclude by providing areas for future work as well as a brief summary of this essay's research.

2 Data

High quality data is essential for good performance of machine learning models and so data extraction played an important part in this project. Applying numerous Structured Query Language (SQL) queries to 5 ParlayPlay databases hosted on Amazon Web Services (AWS) servers, we extracted 34 features from roughly 315 000 users. These features were meant to be representative of a user, with key features including:

- **is_ftd**: a boolean value (True or False) for whether a user has deposited money into their account - known as a First Time Deposit (ftd).
- **net_cash**: (amount withdrawn from) - (amount deposited into) players' accounts. This shows overall how profitable a user was for the company and will be referred to as LTV - or Life Time Value - for the remainder of the essay.
- **contest_count**: total number of contests placed by user. This shows how engaged a user is with the platform.
- **sign_up_time**: when user created account split into 4 categories:
  - **Morning:** 06:00:00 - 11:59:59
  - **Afternoon:** 12:00:00 - 17:59:59
  - **Evening:** 18:00:00 - 23:59:59
  - **Night:** 00:00:00 - 05:59:59

We also wanted to identify the most important *LTV* and *contest_count* values and so we placed these features into intervals, treating them as individual categories. This way, after training our models, we could see precisely which intervals were most influential for the model.

2.1 Data Processing

The data required cleaning and processing before being fed into our machine learning models. Many fields were left empty or as "NaN" values due to users joining the platform but never playing and so were replaced with 0. Furthermore, varying time-zones across the US meant two users could sign up at the same time but have different *sign_up_time* categories (e.g. 11:30 in Los Angeles is 14:30 in New York). To make these consistent, we converted all time zones to Eastern/UTC time.

We also conducted machine learning specific data processes. Categorical data had to be converted to a numerical representation in order to be usable by our machine learning models. We did this through *One Hot Encoding (OHE)* [11] our data. This technique creates a new feature in the data for each unique label in each categorical column. It then places a 1 if the data-point had that label for the respective category and a 0 otherwise. Although this achieves the goal of creating a numerical representation for categorical data, OHE also greatly increases the dimension size of the data whilst encoding very little actual information within. This is a consequence of there being very few 1's - where data is actually encoded - compared to the 0's of remaining columns; this is *sparse* data [12], which can lead to problems with model performance. We discuss mitigation techniques for this later in the essay.

The extreme range in values between features (e.g. a user's LTV could be $150 whilst their average number of picks 2 or 3) could lead to poor training and performance of our models [13]. To avoid this we applied *Min-Max Normalisation* to our numerical data.

Definition 2.1. *(Min-Max Normalisation [14]) Given data X, the Min-Max Normalisation of X is:*

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}.$$

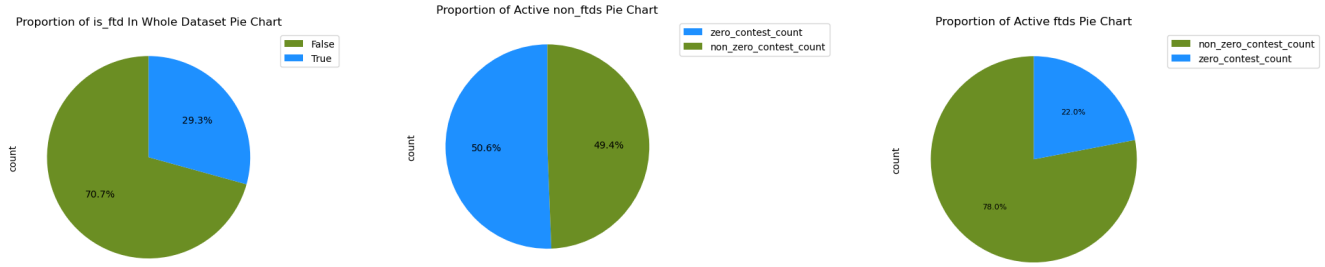
This normalisation technique scales our data to between 0 and 1 while also preserving its shape [15] which was important as initial data analysis indicated a non-Gaussian distribution within our data - see code. As a result, following the processing, our data now had 49 features (increased from 34 by *OHE*) and each data entry was a value between 0 and 1 (scaled by Min-Max Normalisation).

2.2 Exploratory Data Analysis

Prior to employing any machine learning techniques, it's common practice to thoroughly investigate and analyse the dataset on which the model will be trained. This can help with initial pattern and anomaly detection as well as model selection.

A feature key to ParlayPlay's data was whether a user had deposited any money into their account (First Time Deposit - ftd) as this would indicate a commitment to staying and engaging with the platform. The contest_count also served as an additional metric for measuring engagement, as users were only able to place a contest using their own funds. While there were alternative ways to use the platform without personal investment, prioritising contest_count made the most sense for providing a meaningful, business-oriented analysis of engagement. Consequently, users who had not placed any contests were classified as inactive.

Analysis showed that over 70% of all accounts placed no deposit with the platform (Figure 1a) and that, within this group, over 50% (Figure 1b) also placed no contests. This indicated that roughly 35% of all users on the platform were completely inactive. Conversely, of those who did place a deposit, only 22% did not actively engage with the site (Figure 1c). Thus over the whole user base, just over 20% of users placed a deposit and were actively using the platform. From a business standpoint, every user who had placed a deposit held importance, as our aim was to better understand the factors influencing user engagement levels, as well as identifying indicators of users becoming inactive. Consequently, we decided to only perform the cluster analysis on users who had placed a deposit with the platform - totalling roughly 90,000 accounts.



(a) Proportion of ftds in Whole Dataset

(b) Proportion of Active non_ftds

(c) Proportion of Active ftds

Figure 1: Analysis of ftd users

Further analysis can be found in the Appendix. With a clearer understanding of our data, we now proceed to discuss the machine learning techniques used in the project.

3 Machine Learning Background

3.1 KMeans

Our goal was to group similar users to enable an analysis of each group's behaviour; a form of *Unsupervised Learning* known as *Clustering*. There are many different clustering techniques and algorithms but initially, we employed a widely used method: KMeans.

The essence of KMeans clustering is to try to minimise the sum of the squared distances between each point and the centre of its assigned cluster. Formally, given a set $\chi \subseteq \chi'$, where (χ', d) is a metric space with distance d, the centre (also referred to as centroid) of each cluster is defined as follows:

Definition 3.1. *(Cluster Centroid [7])*
The centroid of cluster C_i is

$$\mu_i(C_i) = \arg\min_{\mu \in \chi'} \left(\sum_{x \in C_i} d(x, \mu)^2 \right). \tag{3.1}$$

In machine learning, one often tries to frame a problem in an optimisation light which can then be tackled with appropriate algorithms. This requires defining a function to measure the effectiveness of the model (an objective function) which can then be minimized. For KMeans, our objective function is naturally defined as the sum of the squared distances over each cluster.

Definition 3.2. *(KMeans Objective Function [7])*
For K clusters, the KMeans objective function is:

$$G_{KMeans}((\chi, d), (C_1, ..., C_K)) = \sum_{i=1}^{K} \sum_{x \in C_i} d(x, \mu_i(C_i))^2. \tag{3.2}$$

which can also be written as:

$$G_{KMeans}((\chi, d), (C_1, ..., C_K)) = \min_{\mu_1, ..., \mu_K \in \chi'} \sum_{i=1}^{K} \sum_{x \in C_i} d(x, \mu_i)^2. \tag{3.3}$$

Note 1. *$G_{KMeans}((\chi, d), (C_1, ..., C_K))$ will often be shortened to just $G(C_1, ..., C_K)$*

For our case, we are concerned with minimising distances in $\mathbb{R}^{49}$ as there are 49 features in our data post processing. As a result, we use the Euclidean distance $d(x, y) = \|x - y\|$. We can now discuss the algorithm we will use to minimise this objective function. For a fixed number of clusters, K, we initialise a set of centroids for each cluster and then iteratively update these centroids by finding the average of all the data points assigned to that centroid's cluster. This is repeated until convergence. The number of clusters, K, remains constant throughout the algorithm, making the choice of K particularly important (methods for finding appropriate values of K will be discussed later).

Algorithm 1 KMeans [7]

Require: $\chi \subset \mathbb{R}$, Number of Clusters K

1: **Initialise**: Randomly Choose Initial Centroids $\mu_1, ..., \mu_K$

2: **repeat**

3: Assign each point $x \in \chi$ to the nearest centroid:

$$C_i = \{x \in \chi : \arg\min_j \|x - \mu_j\|\}$$

(Break ties arbitrarily if necessary)

4: Update each centroid μ_i as the mean of the points assigned to it:

$$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

5: **until** Convergence

6: **return** Final cluster assignments and centroids

This algorithm provides a numerical way to get our KMeans clustering and we can prove that it does not increase our objective function.

Lemma 1 ([7])**.** *Each iteration of the KMeans algorithm does not increase the KMeans objective function.*

Proof. Recall our objective function:

$$G(C_1, ..., C_K) = \min_{\mu_1, ..., \mu_K \in \mathbb{R}^n} \sum_{i=1}^{K} \sum_{x \in C_i} \|x - \mu_i\|^2. \tag{3.4}$$

We define:

$$\mu(C_i) = \frac{1}{|C_i|} \sum_{x \in C_i} x \tag{3.5}$$

which is equivalent to

$$\mu(C_i) = \operatorname*{argmin}_{\mu \in \mathbb{R}^n} \sum_{x \in C_i} \|x - \mu\|^2. \tag{3.6}$$

The equivalency is a result of:

$$\begin{aligned}
&\operatorname*{argmin}_{\mu \in \mathbb{R}^n} \sum_{x \in C_i} \|x - \mu\|^2 \\
&= \operatorname*{argmin}_{\mu \in \mathbb{R}^n} \sum_{x \in C_i} < x - \mu, x - \mu > \\
&= \operatorname*{argmin}_{\mu \in \mathbb{R}^n} \sum_{x \in C_i} < x, x > -2 < x, \mu > + < \mu, \mu >
\end{aligned}$$

which, as the argmin is independent of x, can be written as:

$$\begin{aligned}
&= \underset{\mu\in\mathbb{R}^n}{\operatorname{argmin}} |C_i| < \mu, \mu > -2 \sum_{x\in C_i} < x, \mu > \\
&= \underset{\mu\in\mathbb{R}^n}{\operatorname{argmin}} |C_i| < \mu, \mu > -2 < \sum_{x\in C_i} x, \mu > \\
&= \underset{\mu\in\mathbb{R}^n}{\operatorname{argmin}} |C_i| < \mu, \mu > -2|C_i| \frac{1}{|C_i|} < \sum_{x\in C_i} x, \mu > \\
&= \underset{\mu\in\mathbb{R}^n}{\operatorname{argmin}} |C_i| < \mu, \mu > -2|C_i| < \frac{1}{|C_i|} \sum_{x\in C_i} x, \mu > \\
&= \underset{\mu\in\mathbb{R}^n}{\operatorname{argmin}} < \mu, \mu > -2 < \frac{1}{|C_i|} \sum_{x\in C_i} x, \mu > .
\end{aligned}$$

As the argmin depends solely on μ, we could divide through by $|C_i|$ in the last line above. We can also add a term that is solely a function of $\frac{1}{|C_i|}\sum_{x\in C_i}$ to the expression and write:

$$\begin{aligned}
&= \underset{\mu\in\mathbb{R}^n}{\operatorname{argmin}} < \mu, \mu > -2 < \frac{1}{|C_i|} \sum_{x\in C_i} x, \mu > + < \frac{1}{|C_i|} \sum_{x\in C_i} x, \frac{1}{|C_i|} \sum_{x\in C_i} x > \\
&= \underset{\mu\in\mathbb{R}^n}{\operatorname{argmin}} \left\| \mu - \frac{1}{|C_i|} \sum_{x\in C_i} x \right\|^2 .
\end{aligned}$$

We see that the line is minimised by $\mu = \frac{1}{|C_i|}\sum_{x\in C_i} x$ and so (3.5) and (3.6) are equivalent.
We can now re-write our objective function (3.4) as:

$$G(C_1, ..., C_K) = \sum_{i=1}^{K} \sum_{x\in C_i} \|x - \mu(C_i)\|^2. \tag{3.7}$$

We see this by noticing that both give the minimum squared distance between every point and their respective cluster's centre, just taking the minimum in different places. (3.4) minimises the $\mu_1, ..., \mu_k$ over the squared distances from all the points in the dataset whereas (3.7) finds the μ_i's that minimise the squared distances for each cluster and then sums each cluster's combined distance.

We now consider the update after t iterations. Let $C_1^{(t)}, ..., C_K^{(t)}$ be the current partition that minimises objective function with $\mu_i^{(t)} = \mu(C_i^{(t)})$. As the $\mu_i^{(t)}$s minimise the objective function for the partition $C_i^{(t)}$, applying the previous iteration's centroids, $\mu_i^{(t-1)}$, to the objective function (3.4) with the new partition, $C_i^{(t)}$, will increase the objective function, resulting in:

$$\begin{aligned}
G(C_1^{(t)}, ..., C_K^{(t)}) &= \min_{\mu_1^{(t)}, ..., \mu_K^{(t)} \in \mathbb{R}^n} \sum_{i=1}^{K} \sum_{x\in C_i^{(t)}} \|x - \mu_i^{(t)}\|^2 \\
&\leq \sum_{i=1}^{K} \sum_{x\in C_i^{(t)}} \|x - \mu_i^{(t-1)}\|^2.
\end{aligned} \tag{3.8}$$

By definition, our new $C_1^{(t)}, ..., C_K^{(t)}$ minimises $\sum_{i=1}^{K} \sum_{x \in C_i} \|x - \mu_i^{(t-1)}\|^2$ over all possible partitions $(C_1, ..., C_K)$. Therefore,

$$\sum_{i=1}^{K} \sum_{x \in C_i^{(t)}} \|x - \mu_i^{(t-1)}\|^2 \leq \sum_{i=1}^{K} \sum_{x \in C_i^{(t-1)}} \|x - \mu_i^{(t-1)}\|^2. \tag{3.9}$$

We see that the RHS of (3.9) is exactly $G(C_1^{(t-1)}, ..., C_K^{(t-1)})$ whilst the LHS of (3.9) equals the RHS of (3.8). Combining (3.8) and (3.9), we obtain:

$$G(C_1^{(t)}, ..., C_K^{(t)}) \leq G(C_1^{(t-1)}, ..., C_K^{(t-1)}). \tag{3.10}$$

□

In Unsupervised Learning, while ground truth values are not available for the data, there are methods to assess the model's performance. We will employ two standard metrics for evaluating clusters: inertia and silhouette scores. Our aim is to maximise our model's performance with respect to these metrics.

Definition 3.3. *(Inertia [16]) The inertia for a partition $C_1, ..., C_K$ is defined as:*

$$\textit{Inertia} = \sum_{i=1}^{K} \sum_{x \in C_i} d(x, \mu_i)^2. \tag{3.11}$$

For our purposes, $d(x, \mu_i) = \|x - \mu_i\|$.

We see that the objective function for KMeans clustering (3.4) aims to minimise the inertia. Intuitively, inertia measures the compactness of our clusters; lower inertia indicates tighter clusters and signifies a better-performing algorithm.

The second key metric we used to evaluate our clustering was the silhouette score.

Definition 3.4. *(Silhouette Scores [17]) The silhouette score for a given point, $x \in C_i$, is defined as:*

$$S_x = \frac{b_x - a_x}{\max(b_x, a_x)}, \tag{3.12}$$

where

$$a_x = \frac{1}{|C_i| - 1} \sum_{x' \in C_i, x' \neq x} d(x, x') \text{ and } b_x = \min_{j \neq i} \frac{1}{|C_j|} \sum_{x' \in C_j} d(x, x'). \tag{3.13}$$

To get a silhouette score for a partition $C_1, ..., C_K$, we take the average score for all data points.

Intuitively, the silhouette score evaluates how well each data point fits its assigned cluster within our partition. For a point $x \in C_i$, a_x measures the compactness of cluster i with respect to x, while b_x reflects how well x fits to its nearest different cluster (i.e. $b_x = 0$ indicates that the distance between x and its neighboring cluster is 0, suggesting x might belong to this cluster instead). Thus, a silhouette score close to $+1$ signifies lower a_x and higher b_x, indicating compact, well-defined clusters. Conversely, a silhouette score near -1 indicates the opposite scenario. Therefore, for our clustering, we aim for a silhouette score as close to $+1$ as possible.

The KMeans algorithm requires a predefined value of K, and selecting an appropriate value is crucial. Typically, this value is estimated by analysing and evaluating a metric for various values of K to determine the optimal number of clusters. We opted for the "Elbow" Method, a common technique that examines how the inertia changes for different values of K. Initially, as K increases, the inertia drops rapidly before levelling off. This method selects the value of K at which the inertia graph exhibits a noticeable change in slope, thought of as the "elbow" of the graph [18]. It's important to note that recent literature has cautioned against relying solely on the "Elbow" Method, citing its lack of theoretical support [19]. However, despite this criticism, it remains a standard practice in the field and can serve as a useful starting point for analysis. Additionally, we were aware that we would conduct further analysis on each cluster's data post-assignment, allowing us to identify any redundant clusters. Therefore, we made the decision to continue using this method. Figure 2 illustrates the K-inertia graph for KMeans, suggesting that the optimal number of clusters is approximately 15. While the actual number of clusters could potentially be fewer (as indicated by the graph), opting for a higher number provides more opportunities to merge similar clusters without missing important smaller ones that could otherwise be overlooked by the algorithm.

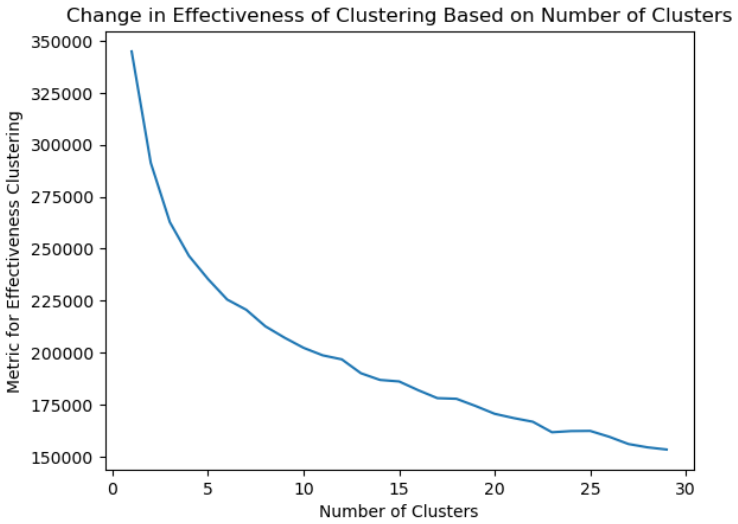


Figure 2: Elbow Graph for KMeans

To implement this clustering, we used a popular machine learning python library, SkLearn [20], which contains the code to run many different machine learning algorithms. We used their "KMeans" class with 15 clusters and 10 runs of randomly initialised centroids, taking the model with the lowest inertia. This model produced an inertia of 186,202.674 and a silhouette score of 0.131. As expected, these indicate very poor clustering and so we searched for a better algorithm.

3.2 Constrained KMeans

One problem with KMeans clustering is that there are no bounds on the size of each cluster, leading to some very small clusters (for example 3 people). This is not overly insightful and so to combat this, we used *Constrained KMeans*: an adaptation of the original algorithm introduced by Microsoft

Research in 2000 [21]. This method allows for a lower bound on the size of each cluster, hopefully leading to better results.

The researchers from Microsoft presented the KMeans objective loss in a different way. For simplicity we let the number of data points be m and the number of clusters be K.

$$G(C_1, ..., C_K) = \min_{\mu_1,...,\mu_K \in \mathbb{R}^n} \sum_{j=1}^{m} \min_{i=1,...,K} \left(\frac{1}{2} \|x_j - \mu_i\|^2 \right). \tag{3.14}$$

Apart from the $\frac{1}{2}$ added to simplify the result after differentiation, this is the same as Equation (3.4). They then replaced the min inside the sum with an indicator function, $T_{i,j}$, that is 1 when point x_j is nearest the centre of cluster i and 0 otherwise. As each point belongs to one and only one cluster, we now have the constraint that for each $j = 1, ..., m$, $\sum_{i=1}^{K} T_{i,j} = 1$. It is proven in [22] that our KMeans problem is then equivalent to:

$$\begin{aligned} &\underset{\mu,T}{\text{minimise}} \sum_{j=1}^{m} \sum_{i=1}^{K} T_{i,j} \cdot \frac{1}{2} \|x_j - \mu_i\|^2 \\ &\text{subject to: for each } j = 1, ..., m, \sum_{i=1}^{K} T_{i,j} = 1 \\ &\text{and for each } i = 1, ..., K \text{ and } j = 1, ..., m, T_{i,j} \geq 0. \end{aligned} \tag{3.15}$$

This problem is solved by the KMeans algorithm and the stationary point found satisfies the Karush-Kuhn-Tucker, KKT, criteria (covered in MA3K1) [21]. However as discussed in [21], this allows for empty clusters. Consequently, the researchers from Microsoft added the additional constraint that each cluster i must have at least τ_i data points (resulting in $\sum_{i=1}^{K} \tau_i \leq m$) and so add to (3.15) that $\sum_{j=1}^{m} T_{i,j} \geq \tau_i$. We can now modify the original KMeans algorithm such that at iteration t, we solve for:

$$\begin{aligned} &\underset{T}{\text{minimise}} \sum_{j=1}^{m} \sum_{i=1}^{K} T_{i,j} \cdot \frac{1}{2} \|x_j - \mu_i^t\|^2 \\ &\text{subject to: for each } i = 1, \ldots, K, \sum_{j=1}^{m} T_{i,j} \geq \tau_i, \\ &\text{for each } j = 1, \ldots, m, \sum_{i=1}^{K} T_{i,j} = 1, \\ &\text{and for each } i = 1, \ldots, K \text{ and } j = 1, \ldots, m, T_{i,j} \geq 0. \end{aligned} \tag{3.16}$$

We then update our non-null cluster centres as the average of all the data points that were assigned to it [21]:

$$\mu_i^{t+1} = \begin{cases} \frac{\sum_{j=1}^{m} T_{i,j}^t x_j}{\sum_{j=1}^{m} T_{i,j}^t} & \sum_{j=1}^{m} T_{i,j}^t \geq 0 \\ \mu_i^t & \text{otherwise} \end{cases}.$$

We used the "K-means-constrained" python package implemented by the Microsoft researchers and set the minimum cluster size to 200. This threshold was chosen to ensure analysis drawn from the clusters would provide meaningful insights into genuine user trends, rather than being influenced by outlier. Through the "Elbow Method" and Figure 3, we decided on 15 clusters.

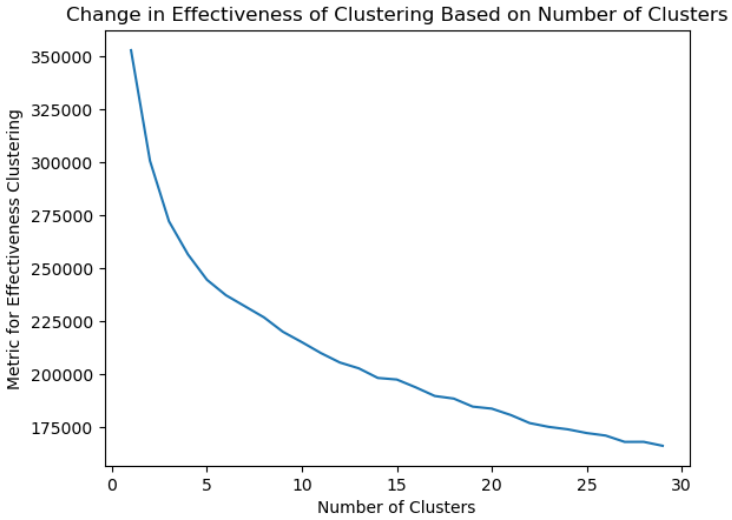


Figure 3: Elbow Graph for Constrained KMeans

This led to much better, more general clusters from which meaningful insights could be made. There were limitations with this algorithm however. Normal KMeans has a time complexity of $O(mK)$ where m in the number of data points and K the number of clusters [23]. The Constrained KMeans algorithm on the other hand has a larger time complexity of $O((m^3K + m^2K + mK)\log(m + K))$ [23]. This meant the Constrained KMeans algorithm took significantly longer to run: the normal KMeans took roughly 15 seconds whilst for the same conditions (e.g. same data, number of clusters and number of iterations), the constrained version took almost half an hour (28 minutes 28 seconds). That said, it did lead to better clustering. After 10 runs of randomly initialised centres with 15 clusters and taking the model with the lowest inertia, our Constrained KMeans model produced an inertia of 185,787.599 and a silhouette score of 0.144. This is roughly a 0.2% decrease in inertia and a 10% increase in silhouette score compared with the normal KMeans algorithm. Both of these metrics indicate improved clustering, however we still felt improvements could be made.

4 Dimensionality Reduction

4.1 PCA Reduction

As discussed during Data Processing, there are mitigation techniques to overcome the problems associated with high dimensional sparse data caused by One Hot Encoding. One such method is Principal Component Analysis (PCA) reduction that uses singular value decomposition (SVD) to represent our data in lower dimensions [24]. Taking our data as a matrix $X \in \mathbb{R}^{n \times p}$ where n is the number of data points and p the number of features, we standardise our values by subtracting the

mean of each column and dividing by the sum of squares errors to make X_s [24]:

$$\begin{aligned} \text{Mean of each column: } \bar{x}_j &= \frac{1}{n}\sum_{i=1}^{n} X_{i,j} \ , \\ \text{Sum of Squares Errors per column: } \mathrm{s}_j^2 &= \sum_{i=1}^{n}(x_{ij}-\bar{x}_j)^2, \\ \text{Then } X_s \text{ is: } (X_s)_{ij} &= \frac{x_{ij}-\bar{x}_j}{s_j}. \end{aligned} \tag{4.1}$$

With n large ($\sim 10^8$), this standardisation becomes impractical ; however, as our data is a manageable size, this does not present a problem here. We assume $\text{rank}(X) = \text{rank}(X_s) = r < p$ and we take the SVD of $X_s = U\Sigma V^T$, with U and V orthogonal $n \times r$ and $p \times r$ respectively matrices with $U^TU = V^TV = I_r$ and $\Sigma = \text{diag}(\sigma_1, \sigma_2, ..., \sigma_r)$ an $r \times r$ matrix with ordered singular values along its main diagonal [24] (recall from MA398 that the singular values are the square roots the eigenvalues of $X_s^TX_s$). The *Principal Components (PCs)* are then the columns of $U\Sigma$ whilst the columns of V are the *loadings* (or weights for each feature). For a given principal component j, its sample (or explained) variance is given by σ_j^2/n. We can then reduce our dimension size by picking a value, $w < r$ and truncating our matrices to these sizes [24]:

$$X_{w,s} = \sum_{i=1}^{w} \sigma_j u_j v_j^T = U_w \Sigma_w V_w^T. \tag{4.2}$$

The total variation of this truncated matrix is therefore $\sum_{j=1}^{w} \sigma_j^2/n$ which can be shown as a proportion of the total variance [24]:

$$\lambda_w = \frac{\sum_{i=1}^{w} \sigma_j u_j v_j^T}{\sum_{i=1}^{r} \sigma_j u_j v_j^T}. \tag{4.3}$$

We want to find an optimal value of w that sufficiently reduces the dimensionality of the data but also has $\lambda_w \approx 1$ (i.e. most of X_s's information is encoded in $X_{w,s}$).

We implemented this reduction using SkLearn's PCA decomposition class and analysed the explained variance of each principal component. We found that taking the first 10 components resulted in a total explained variance of 0.74 (Figure 4a). Whilst not perfect, this variance exceeds the commonly accepted variance threshold of 0.7 for a good reduction [25] and since adding more components did not greatly increase the variance, we decided to only use the first 10 components. Additionally, this SkLearn class also included an attribute that returned the features corresponding to maximum variance in the data. As a result, we could find each principal component's most influential columns (Figure 4b), giving deeper insights into not only the workings of this method but also the data as a whole.

Explained Variance for Number of Dimensions:

	0
0	0.164718
1	0.096030
2	0.091317
3	0.082167
4	0.073470
5	0.064511
6	0.053786
7	0.043417
8	0.038823
9	0.034677
10	0.032585
11	0.031571
12	0.029914
13	0.025987
14	0.023094

Sum of variance for first 10 dimensions:
0.7429157678919406

(a) Explained Variance Per Principal Component

Each PC's top 3 most influential columns:

	0
0	[result_0, contest_count_binned_(-1, 0], result_lost]
1	[contest_count_binned_(10, 50], ltv_binned_(0.0, 25.0], result_won]
2	[sign_up_time_afternoon, sign_up_time_evening, sign_up_time_morning]
3	[coupon_designation_Unattributed, result_won, result_lost]
4	[coupon_designation_Unattributed, ltv_binned_(0.0, 25.0], result_won]
5	[sign_up_time_morning, sign_up_time_evening, sign_up_time_afternoon]
6	[contest_count_binned_(5, 10], contest_count_binned_(10, 50], ltv_binned_(0.0, 25.0]]
7	[contest_count_binned_(0, 5], ltv_binned_(-1000000.0, -0.1], contest_count_binned_(5, 10]]
8	[ltv_binned_(-1000000.0, -0.1], ltv_binned_(100.0, 1000.0], ltv_binned_(50.0, 100.0]]
9	[ltv_binned_(50.0, 100.0], ltv_binned_(100.0, 1000.0], contest_count_binned_(0, 5]]

(b) Top 10 Principal Component's Most Influential Columns

Figure 4: Implementation of PCA Reduction

We then clustered the 10 component representation of our data using normal KMeans with 15 clusters and 10 randomly initialised centres and found a best inertia score of 101,296.187 with a silhouette score 0.226. This is roughly a 40% decrease in inertia and a 40% increase in silhouette score compared with the Constrained KMeans algorithm. This indicates a far better clustering than just normal KMeans and Constrained KMeans, showcasing the benefit of reducing dimensionality.

4.2 Deep Clustering

The total explained variance of 0.74 achieved in our PCA reduction (Figure 4a) exceeds the commonly accepted cutoff of 0.7 [25], indicating a good-quality PCA reduction. Given this technique is a linear decomposition, this implies our data may exhibit underlying linear properties. That said, the total explained variance is not perfect (the ideal being 1) which indicates there could be benefit in exploring non-linear reduction techniques. One such technique uses *AutoEncoders* [26] which are the basis of state-of-the-art deep clustering [1]. Before discussing this approach, however, we briefly explore some important terms in deep learning.

As covered in the Introduction, deep learning involves multiple neural networks (NNs), each containing a non-linear *activation function* which decides when the neuron should fire (discussed in MA3K1). Initially the data is stored in an *input layer* which is then fed into a series of NNs, arriving at a *hidden layer* after flowing through each of these networks. Eventually the data is returned at the output layer. This is depicted in Figure 5. One can also choose to randomly turn off neurons in the network (i.e. not let any data through this neuron), known as *Dropout*, which helps prevent the network over-relying on certain neurons. This leads to better, more robust models [27].

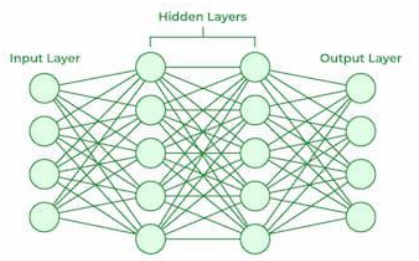


Figure 5: Neural Networks [28]

AutoEncoders are a class of NNs that learn to reconstruct the model's input from a smaller representation. As seen in Figure 6, the data inputted from the LHS is reduced in size by the blue neural network (known as the *Encoder*) to create a smaller representation of the data (the red layer, known as the *Latent Space* or sometimes *Embedding*). This smaller representation is then fed into the green neural network (known as the *Decoder*) which attempts to recreate the original data first fed into the blue neural network.

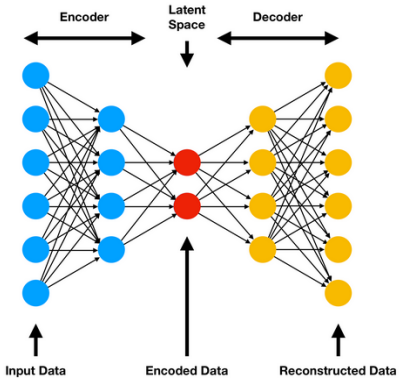


Figure 6: AutoEncoder Architecture [29]

The input and output are then compared using a loss function [26]. For our case, we used the *Mean Squared Error* (MSE) loss [30].

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{input} - y_{output})^2. \tag{4.4}$$

This loss is then used to update the model's parameters through an optimisation algorithm. This update can occur for each data point in the training data (data set aside specifically for training purposes) but this can be time consuming and costly - especially for large quantities of data. Consequently, one splits the data into *batches* of a certain pre-determined size and loops through these, updating the model's parameters based on all the data points for each batch. One full sweep through the whole data (either individually or through each batch) is known as an epoch.

In MA3K1 we discussed various optimisation algorithms commonly used in machine learning such as Gradient Descent (GD) and Stochastic Gradient Descent (SGD). However, these algorithms can be slow [31] and so we used two different optimisation techniques which have faster convergence rates than SGD: *Adam* [32] and *SGD with Momentum* [33]. This essay will discuss SGD with Momentum in detail, proving its convergence for smooth, convex functions as well as briefly covering Adam.

Before discussing SGD with Momentum, it is worth mentioning *Exponentially Weighted Averages* as these are key to this optimisation algorithm.

Definition 4.1. *(Exponentially Weighted Average (EWA) [34]) The EWA, v_k, at step k for data $\theta_1, ..., \theta_n$ and weighting β is defined as:*

$$v_k = \beta v_{k-1} + (1 - \beta)\theta_k. \tag{4.5}$$

Note 2. *It is worth highlighting here that the k used throughout this section refers to the k^{th} iterate of the respective algorithm and is different to the K found in previous sections which represents the number of clusters.*

v_k is often initialised to 0 or the mean of the data [34]. The EWA is a way of calculating a moving average that "remembers" the previous data and the value of β helps smooth this moving average. This can be applied to optimisation when one wants the algorithm to "remember" the previous weights.

SGD with Momentum extends SGD by incorporating exponentially weighted averages to the update step.

Definition 4.2. *(SGD With Momentum [35]) Let $f = \frac{1}{n}\sum_{i=1}^{n} f_i(x)$ be the function we wish to minimise and let w_k be the model's parameters (as defined in MA3K1). Then for SGD with Momentum, the weights are updated according to:*

$$w_{k+1} = w_k - \eta v_k$$

$$\text{where } v_k = \beta v_{k-1} + (1 - \beta)\nabla f_i(w_k).$$

Note 3. *Often in literature the update step is scaled to remove the $(1 - \beta)$ term, which will be the convention for the rest of the essay. Note that this can cause problems when tuning β as one may have to also re-tune η.*

An interpretation for this new algorithm is considering a ball moving on a non-convex surface. The new term v_k can be thought of as the momentum of the ball at a given time k. Imagine dropping this ball down a slope into a local minima. Given a high enough momentum, the ball will hopefully travel out of this minima and continue to roll, looking for the global minimum. This algorithm can be thought of as giving each iteration enough momentum to not get stuck in local minima or saddle points [36].

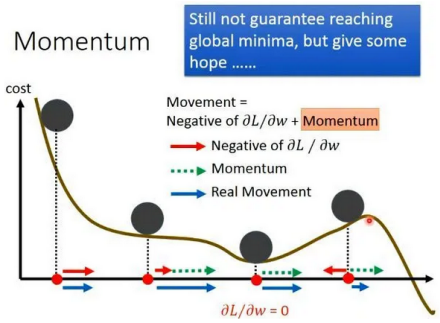


Figure 7: SGD with Momentum Ball Intuition [37]

Another way of writing SGD with Momentum that will be useful later is discussed in the Lemma below.

Lemma 2. *([35]) The SGD with Momentum defined in Note 2, Definition 9.2, can be re-written as:*

$$w_{k+1} = w_k - \eta \nabla f_i(w_k) + \beta(w_k - w_{k-1}).$$

Proof. Combining the two equations, using the form from Note 2:

$$w_{k+1} = w_k - \eta(\beta v_{k-1} + \nabla f_i(w_k)).$$

Then just using the formula for v_{k-1}. □

We can also write the SGD with Momentum algorithm as an *Iterative Moving Average (IMA).*

Lemma 3 (Iterative Moving Algorithm [35])**.** *We introduce a new parameter, z_t, at each step $t \in \mathbb{N}$ with $z_{-1} = 0$ and re-write the SGD update line as follows:*

$$z_t = z_{t-1} - \eta_t \nabla f_{it}(w_t)$$

$$w_{t+1} = \frac{\lambda_{t+1}}{\lambda_{t+1} + 1} w_t + \frac{1}{\lambda_{t+1} + 1} z_t.$$

Additionally, if $(w_t)_{t \in \mathbb{N}}$ is generated by the SGD with Momentum algorithm with parameters, γ_t, β_t, i.e.

$$w_{t+1} = w_t - \gamma_t f_i(w_t) + \beta_t(w_t - w_{t-1}),$$

then this algorithm satisfies the IMA version with parameters:

$$\beta_t = \frac{\gamma_{t-1}\lambda_t}{\gamma_t(1 + \gamma_{t+1})},\ \eta_t = (1 + \lambda_{t+1})\gamma_t,\ \text{and}\ z_{t-1} = w_t + \gamma_t(w_t - w_{t-1}).$$

Proof. The proof is omitted for brevity but can be found in [35]. □

Before proving SGD with Momentum's convergence for smooth, convex functions, we need to discuss two assumptions that will be used in the theorem. Note the following results for this section were taken from [35].

We first define $L - Smooth$ functions which are key to one of the assumptions.

Definition 4.3 (L-Smooth)**.** *We say that $f : \mathbb{R}^n \to \mathbb{R}$ is L-Smooth for $L > 0$, if it is differentiable and $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ is L-Lipschitz, i.e.:*

$$\forall x, y \in \mathbb{R}^n,\ \|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|.$$

Recall we are trying to minimise a function $f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x)$. We now make the following assumptions:

Assumption 1 (Sum of L_{max} Smooth). *The function f is Sum of L_{max} Smooth if each of the f_i's are L_i Smooth. We take $L_{max} = \max_{i=1,\dots n} L_i$*

Assumption 2 (Sum of Convex)**.** *The function f is Sum of Convex if each of the f_i's are convex.*

We can now state the theorem that SGD with Momentum converges for smooth, convex functions.

Theorem 1. *Let Assumptions 1 and 2 hold. Consider $(w_t)_{t\in\mathbb{N}}$ as iterations in the SGD with Momentum algorithm with parameters:*

$$\gamma_t = \frac{2\eta}{t+3},\ \beta_t = \frac{t}{t+2}\ \text{with}\ \eta \leq \frac{1}{4L_{max}},$$

then the iterates converge according to

$$\mathbb{E}[f(w_t) - \inf f] \leq \frac{\|w_0 - w_*\|^2}{\eta(t+1)} + 2\sigma_f^*.$$

Note 4. *σ_f^* is the Gradient Noise and will be defined later.*

Before proving Theorem 1, we first discuss a useful result due to the convexity of f.

Lemma 4. *If $f : \mathbb{R}^2 \to \mathbb{R}$ is convex and differentiable, then*

$$\forall x, y \in \mathbb{R}^n,\ f(x) \geq f(y) + \langle \nabla f(x), x - y\rangle.$$

Proof. By convexity of f, we have

$$\frac{f(x + \lambda(x - y)) - f(y)}{\lambda} \leq f(x) - f(y).$$

We then get, by taking the limit as $\lambda \to 0$ $\langle \nabla f(x), x - y\rangle \leq f(x) - f(y)$ $\square$

Recall σ_f^* from Theorem 1, this is known as the *Gradient Noise*:

Definition 4.4 (Gradient Noise)**.** *Assume f holds our Sum of L_{max} Smooth assumption. We define:*

$$\sigma_f^* = \inf_{x_* \in \operatorname{argmin} f} \mathbb{V}[\nabla f_i(x_*)]$$

where, for a given random variable, X,

$$\mathbb{V}[X] = \mathbb{E}[\|X - \mathbb{E}[X]\|^2].$$

The *Gradient Noise* relates to *interpolation.*

Definition 4.5 (Interpolation)**.** *Interpolation is said to hold when there exist* $x_* \in \mathbb{R}^n$ *such that* $f_i(x_*) = \inf f_i$ *for all* $i = 1, ..., n$.

As a result, we see that when interpolation holds, $\sigma_f^* = 0$. Since $\sigma_f^* \geq 0$, the Gradient Noise reflects how close we are to interpolation with respect to the gradients of our functions.

The Gradient Noise leads into another important lemma that we will need to prove Theorem 1.

Lemma 5 (Variance Transfer)**.** *If Assumptions 1 and 2 hold for a function f, then for all* $x \in \mathbb{R}^n$

$$\mathbb{E}[\|\nabla f_i(x)\|^2] \leq 4L_{max}(f(x) - \inf f) + 2\sigma_f^*.$$

This lemma shows us where σ_f^* comes from in Theorem 1. In order to prove Lemma 5, we need a few more results.

Lemma 6. *If* $f : \mathbb{R}^n \to \mathbb{R}$ *is convex and L-smooth then for all* $x, y \in \mathbb{R}^n$,

$$\frac{1}{2L}\|\nabla f(y) - \nabla f(x)\|^2 \leq f(y) - f(x) - \langle \nabla f(x), x - y \rangle.$$

Proof. By the convexity and smoothness of f, Lemma 4 and for any $z \in \mathbb{R}^n$,

$$f(x) - f(y) = f(x) - f(z) + f(z) - f(y)$$

$$\leq \langle \nabla f(x), x - z \rangle + \langle \nabla f(y), y - z \rangle + \frac{L}{2}\|z - y\|^2.$$

Minimising the RHS w.r.t. z gives

$$z = y - \frac{1}{L}(\nabla f(y) - \nabla f(x)).$$

Substituting in z and re-arranging gives the result:

$$f(x) - f(y) \leq \langle \nabla f(x), x - z \rangle + \langle f(y), y - z \rangle + \frac{L}{2}\|z - y\|^2$$

$$= \langle \nabla f(x), x - y \rangle - \frac{1}{2L}\|f(y) - \nabla f(x)\|^2.$$

□

Lemma 7. *If Assumptions 1 and 2 hold, then f is* L_{max} *Smooth in expectation, i.e. for all* $x, y \in \mathbb{R}^n$

$$\frac{1}{2L_{max}}E[\|\nabla f_i(y) - \nabla f_i(x)\|^2] \leq f(y) - f(x) - \langle \nabla f(x), y - x \rangle.$$

Proof. Applying Lemma 6 to f_i and using that $L_i \leq L_{max}$, we can write:

$$\frac{1}{2L}\|\nabla f_i(y) - \nabla f_i(x)\|^2 \leq f_i(y) - f_i(x) + \langle \nabla f_i(x), y - x \rangle.$$

Multiplying through by $\frac{1}{n}$ and summing over i gives the result. □

We can then take Lemma 7 further.

Lemma 8. *If Assumptions 1 and 2 hold, then for any $x \in \mathbb{R}^n$ and $x_* \in \operatorname{argmin} f$, we have*

$$\frac{1}{2L_{max}}\mathbb{E}[\|\nabla f_i(x) - \nabla f_i(x_*)\|^2] \leq f(x) - \inf f.$$

Proof. Apply Lemma 7 with $x = x_*$ and $y = x$. The result follows as $f(x_*) = \inf f$ and $\nabla f(x_*) = 0$. □

The final Lemma we need concerns the Gradient Noise.

Lemma 9. *If Assumptions 1 and 2 hold, then $\sigma_f^* = \mathbb{V}[\nabla f_i(x_*)]$ for every $x_* \in \operatorname{argmin} f$.*

Proof. Let $x_*, x' \in \operatorname{argmin} f$, we want to show that $\mathbb{V}[\nabla f_i(x_*)] = \mathbb{V}[\nabla f_i(x')]$. From Lemma 8, we have:

$$\frac{1}{2L_{max}}\mathbb{E}[\|\nabla f_i(x') - \nabla f_i(x_*)\|^2] \leq f(x') - \inf f = \inf f - \inf f = 0.$$

Consequently, we see that

$$\mathbb{E}[\|\nabla f_i(x') - \nabla f_i(x_*)\|^2] = 0$$

and so for every $i = 1, ..., n$, we have $\|\nabla f_i(x') - \nabla f_i(x_*)\| = 0$, i.e. that $\nabla f_i(x') - \nabla f_i(x_*) = 0$. Recall that $\mathbb{V}[X] = \mathbb{E}[\|X - \mathbb{E}[X]\|^2]$ and so with $\nabla f_i(x') = \nabla f_i(x_*)$, we therefore have $\mathbb{V}[\nabla f_i(x_*)] = \mathbb{V}[\nabla f_i(x')]$. □

We can now use these results to prove the Variance Transfer Lemma (Lemma 5).

Proof. (Lemma 5) Let $x_* \in \operatorname{argmin} f$, then $\nabla f_i(x) = 0$ so $\sigma_f^* = \mathbb{V}[\|\nabla f_i(x_*)\|^2]$ from Lemma 9. We then use:

$$\|\nabla f_i(x)\|^2 \leq 2\|\nabla f_i(x) - \nabla f_i(x_*)\|^2 + 2\|\nabla f_i(x_*)\|^2$$

and take the expectation over this. Applying Lemma 8 yields the result. □

We can now prove that *SGD with Momentum* converges for smooth, convex functions (Theorem 1).

Proof. (Theorem 1) Using Lemma 3, we can take the Iterative Moving Average form of the *SGD with Momentum* algorithm and use parameters:

$$\eta_t = \eta,\ \gamma_t = \frac{t}{2},\ \text{and}\ z_{t-1} = w_t + \gamma_t(w_t - w_{t-1}).$$

We then consider iterates, (w_t, z_t), of the IMA algorithm. We start by studying $\|z_t - w_*\|^2$:

$$\|z_t - w_*\|^2 = \|z_{t-1} - w_* - \eta\nabla f_i(w_t)\|^2$$

$$= \|z_{t-1} - w_*\|^2 + 2\eta\langle\nabla f_{it}(w_t), w_* - z_{t-1}\rangle + \eta^2\|\nabla f_{it}(w_t)\|^2.$$

Then from the above equation for z_{t-1}:

$$= \|z_{t-1} - w_*\|^2 + 2\eta\langle\nabla f_{it}(w_t), w_* - w_t\rangle + 2\eta\gamma_t\langle\nabla f_{it}(w_t), w_{t-1} - w_t\rangle + \eta^2\|\nabla f_{it}(w_t)\|^2.$$

For $t = 0$, we take $w_{-1} = 0$. Then from Lemma 4, noting that for $x_* \in \operatorname{argmin} f$, $f(x_*) = \inf f$, we have:

$$\mathbb{E}[\|z_t - w_*\|^2|w_t] = \|z_{t-1} - w_*\|^2 + 2\eta\langle\nabla f(w_t), w_* - w_t\rangle + 2\eta\gamma_t\langle\nabla f(w_t), w_{t-1} - w_t\rangle + \eta^2\mathbb{E}_t[\|\nabla f(w_t)\|^2|w_t]$$

$$\leq \|z_{t-1} - w_*\|^2 + 2\eta(\inf f - f(w_t)) + 2\eta\gamma_t(f(w_{t-1}) - f(w_t)) + \eta^2\mathbb{E}_t[\|\nabla f(w_t)\|^2|w_t].$$

From the Variance Transfer Lemma we then also have:

$$\mathbb{E}[\|z_t - w_*\|^2 | w_t] \leq \|z_{t-1} - w_*\|^2 + 2\eta(\inf f - f(w_t)) + 2\eta\gamma_t(f(w_{t-1}) - f(w_t)) + 4\eta^2 L_{max}(f(w_t) - \inf f) + 2\eta^2\sigma_f^*$$

$$= \|z_{t-1} - w_*\|^2 + 2\eta\gamma_t(f(w_{t-1}) - f(w_t)) + (4\eta^2 L_{max} - 2\eta)(f(w_t) - \inf f) + 2\eta^2\sigma_f^*$$

$$= \|z_{t-1} - w_*\|^2 - 2\eta(1 + \gamma_t - 2\eta L_{max})(f(w_t) - \inf f) + 2\eta\gamma_t(f(w_{t-1}) - \inf f) + 2\eta^2\sigma_f^*$$

$$\leq \|z_{t-1} - w_*\|^2 - 2\eta\gamma_{t+1}(f(w_t) - \inf f) + 2\eta\gamma_t(f(w_{t-1}) - \inf f) + 2\eta^2\sigma_f^*.$$

The last inequality used that $\eta \leq \frac{1}{4L_{max}}$ and that $\gamma_{t+1} = \gamma_t + \frac{1}{2}$. We now take the expectation and sum from $t = 0, ..., T$.

$$\mathbb{E}[\|z_T - w_*\|^2] \leq \|z_{-1} - w_*\|^2 - 2\eta\gamma_{T+1}\mathbb{E}[f(w_T) - \inf f] + 2\eta\gamma_0(f(w_{-1}) - \inf f) + 2\eta^2\sigma_f^*(T+1).$$

With $\gamma_0 = 0$, we have $z_{-1} = w_0 + \gamma_0(w_0 - w_{-1}) = w_0$. Note also that $0 \leq \mathbb{E}[\|z_T - w_*\|^2]$. Combining these results, we have:

$$2\eta\gamma_{T+1}\mathbb{E}[f(w_T) - \inf f] \leq \|w_0 - w_*\|^2 + 2(T+1)\eta^2\sigma_f^*.$$

We now divide through by $2\eta\gamma_{T+1}$, recalling from the parameters at the start of the proof that $2\gamma_{T+1} = T + 1$. This results in Theorem 1.

$$\mathbb{E}[f(w_T) - \inf f] \leq \frac{\|w_0 - w_*\|^2}{\eta(T+1)} + 2\sigma_f^*.$$

□

Through specific choices of η and T, we can bound $\mathbb{E}[f(w_T) - \inf f]$ by any $\epsilon > 0$.

Corollary 1. *For any $\epsilon > 0$, we can bound $\mathbb{E}[f(w_T) - \inf f] < \epsilon$ by taking:*

$$\eta = \frac{1}{\sqrt{T+1}}\frac{1}{4L_{max}}, \text{ and } T \geq \frac{1}{\epsilon^2}\frac{1}{4L_{max}^2}(\frac{1}{2}\|w_0 - w_*\|^2 + \sigma_f^*)^2.$$

Proof. Plug this value of η into Theorem 1 and one gets:

$$\mathbb{E}[f(w_T) - \inf f] \leq \frac{1}{2L_{max}}\frac{1}{\sqrt{T+1}}(\frac{1}{2}\|w_0 - w_*\|^2 + \sigma_f^*).$$

We then take T such that the above is less than ϵ. □

We have now proven the convergence of SGD with Momentum for smooth, convex functions. This section was based on [35], where results on non-smooth functions as well as other stochastic gradient methods can also be found.

As well as SGD with Momentum, we also used *Adam*. This optimisation algorithm combines elements of momentum (discussed above) and adaptive learning rates (where the algorithm's learning rate is updated after each iteration) [38]. Specifically Adam is partly based on *Root Mean Square Propagation* (RMSProp) which uses a moving average over the squared gradients to help update the model's weights.

Definition 4.6 (RMSProp Update Step [39])**.** *For a model with weights, w_k, we define the update step with $\gamma, \eta \in \mathbb{R}, \epsilon > 0$:*

$$w_{k+1} = w_k - \frac{\eta}{\sqrt{\mathbb{E}[(\nabla f_i(w_k))^2]_k + \epsilon}} \nabla f_i(w_k) \tag{4.6}$$

with

$$\mathbb{E}[(\nabla f_i(w_k))^2]_k = \gamma \mathbb{E}[(\nabla f_i(w_k))^2]_{k-1} + (1-\gamma)(\nabla f_i(w_k))_k^2. \tag{4.7}$$

Adam has consistently demonstrated a fast rate of convergence across a range of different tasks [32] which is why we chose this optimisation algorithm for pre-training (discussed below) our autoencoders. That said, literature shows that models trained using SGD based algorithms generalise better than ones trained using Adam [40]. Generalisation refers to the model's ability to adapt to and perform well on data not used during the training process (unseen data) [41]. In other words, the model learns broadly what is important for its task rather than relying on features specific to the data used in training. This leads to better, more robust models and as a result, we, along with existing deep clustering literature [1], used SGD with Momentum with a momentum of 0.9 for model fine-tuning (also discussed below).

Before discussing the deep clustering architecture, we briefly cover two key concepts of the theory. Often in deep learning, one trains a model to learn a rough idea of the landscape of the problem, known as *pre-training*, and then re-trains the model again on task specific data to hone the model's ability to perform this task, known as *fine-tuning*. An analogy for this could be screwing a screw to join two planks of wood: the pre-training refers to screwing down until the two planks hold loosely together but then fine-tuning, screws even further until the planks hold tightly. The key idea behind deep clustering is that one pre-trains the autoencoder before fine-tuning the encoder part of the model.

First introduced in 2016, *Deep Embedded Clustering* (DEC) [1] revolutionised the clustering landscape with a novel approach based on deep learning. The researchers proposed initially feeding the data through a pre-trained autoencoder's encoder to give a dimensionally reduced representation of the data (thought of as a deep embedding for the data). A KMeans clustering, with a pre-determined number of clusters, is then performed on these representations, resulting in an initial centre for each cluster. One then takes these representations, together with each cluster's centre, to form a probability distribution over the clusters. This distribution is then used to update the encoder's parameters. The original data now passes through the new encoder to give updated representations and the process is repeated.

The probability distribution over the clusters is created using the *Student's t-Distribution*.

Definition 4.7. *(Student's t-Distribution [1]) Let the representation of a data point be z_i and a cluster centre μ_j, define:*

$$q_{ij} = \frac{(1 + \|z_i - \mu_j\|^2)^{-1}}{\sum_{j'}(1 + \|z_i - \mu_{j'}\|^2)^{-1}}.$$

q_{ij} can be interpreted as the probability of point z_i belonging to cluster μ_j. The matrix $Q = (q_{ij})_{n \times K}$ then stores these probabilities for n data points and K clusters.

Note 5. *In literature, there is a term α in the equation that represents the degrees of freedom of the Student's t-Distribution. It has been shown that learning α is unnecessary and cross-validation of α on a validation set is not possible [1]. As with the rest of research using this distribution for deep clustering, we took $\alpha = 1$.*

We observe that as $z_i \to \mu_j$, the numerator tends to 1, whilst the further away a point is, the nearer the numerator is to 0. The denominator then makes all the values for a specific i sum to 1 (i.e. the sum of the probabilities that point i belongs to each cluster is 1). We therefore have a probability distribution where the closer a point is to the centre of a cluster, the higher the probability this point belongs to that cluster.

In order to update the encoder's parameters, we need a target or "ideal" distribution to which we can compare the one above. The researchers in DEC proposed basing this target distribution on the q_{ij}.

Definition 4.8. *(DEC's Target Distribution [1]) Let p_{ij} be the ideal probability of point z_i belonging to cluster μ_j. We define p_{ij} in the following way:*

$$p_{ij} = \frac{q_{ij}^2/f_j}{\sum_{j'} q_{ij'}^2/f_{j'}}$$

$$\text{where } f_j = \sum_i q_{ij}.$$

Note 6. *To avoid confusion, the f_j defined here represent the normalisation of cluster j and are different to the functions f_i used when discussing SGD With Momentum.*

This distribution looks to improve cluster purity whilst also preventing large clusters from significantly impacting the representations through the normalisation term, f_j [1]. We can now create a matrix $P = (p_{ij})_{n \times K}$ as a matrix of the ideal probabilities of point i belonging to cluster j.

We now compare these two distributions using the Kullback-Leibler (KL) loss.

Definition 4.9. *(Kullback-Leibler (KL) Loss [42]) For probability distributions, P and Q, the KL loss is defined as:*

$$L = KL(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}.$$

This loss is then used to update the parameters of the encoder. KL loss is discussed in MA3K1 and so will not be discussed further here.

The autoencoder pre-trained in DEC had 8 hidden layers of sizes 500, 500, 2000, 10, 2000, 500, 500 respectively. They also used ReLU (defined below) as the activation function on all layers besides those into the latent space (the 2000 to 10) and the output (the last 500 to 500). These two layers create representations of the original data that are used later in the algorithm; consequently, at these layers, all the data needs to be present and not stopped by the activation function.

Definition 4.10. *(ReLU [42]) The Rectified Linear Unit (ReLU) is defined as follows:*

$$ReLU(x) = \max(0, x). \tag{4.8}$$

There were various other deep learning specific setups for this model (e.g. batch size and dropout rate) that can be found in the original paper [1]. Ultimately, the total training for DEC, including pre-training and fine-tuning, resulted in 500,000 epochs producing a truly state-of-the-art clustering model. Training a model this big and for this amount of time was not only impractical for the resources available to us but also potentially not appropriate either. Excessive training can lead to the model solely learning the training data and not generalising well when working with unseen data (known as *overfitting*). Given we had roughly 90,000 data points to train a model with over 2 million parameters (see MA3K1 for this calculation), training for half a million epochs would almost certainly lead to overfitting. As a result, we decided to use a scaled down version of DEC: using an autoencoder with 3 hidden layers of sizes 49, 10, 49 respectively (5,782 parameters in total - see MA3K1 for this calculation). It is worth recalling that our data was in $\mathbb{R}^{49}$ so we decided to not change dimension size into the first or out of the last hidden layers. Additionally, PCA reduction decreased our data's dimension size to 10 and so to directly compare our deep clustering to our PCA reduction, we chose to reduce the dimension size to 10 here as well. Following DEC, we also used ReLU on all hidden layers besides those into the latent space and output. We also set the batch size to 64 and had a dropout rate of 0.5 (i.e. 50% of neurons are switched off randomly); although this may seem a high rate, it is commonly used in deep learning. In all, we trained for 200 epochs (100 for the pre-training and fine-tuning each) as well as implementing *early stopping* where the model stops training once it starts showing signs of overfitting (i.e. when the model's performance on unseen data plateaus or worsens after three consecutive epochs of training).

We trained (pre-trained and fine-tuned) 5 separate models as running 10 times, like with the KMeans models, would have taken too long and took the median of all the results. We took the median as occasional poor performances would have greatly impacted the final score for this model had we taken the average. The pre-trained encoder before fine-tuning resulted in an inertia of 2605.904 and a silhouette score of 0.501. That is a 97% decrease in inertia from the PCA reduction and over a 120% increase in silhouette score which indicates a much higher quality of clustering. After fine-tuning, however, we had an inertia of 0.768 but a silhouette score of 0.416. Despite the near perfect inertia, fine-tuning resulted in a 17% decrease in silhouette score indicating worse clustering, contrary to our expectations. Despite this disappointment, we decided to analyse the algorithm and see if any improvements could be made.

We identified that the target distribution, P, could be adapted to better suit our needs. In DEC's original paper, the researchers dismiss using the *Nearest Cluster Delta Distribution* (defined below), explaining it to be naive and too simplistic. Given the size of our data and model, we believed a simpler distribution could lead to better results and so looked to implement this.

Definition 4.11. *(Nearest Cluster Delta Distribution [43]) Let the representation for a point be z_i. The Nearest Cluster Delta Distribution is defined as follows:*

$$p_{ij} = \begin{cases} 1 & \text{if } z_i \text{ is closest to cluster } j \\ 0 & \text{otherwise} \end{cases}.$$

We found that this distribution resulted in an inertia of 0.040 and a silhouette score of 0.648. Compared to DEC's standard distribution, this is a decrease of 95% in inertia and an increase of over 50% in silhouette score. This approach produced the most successful clustering results thus far, prompting us to recognise the potential of the Delta Distribution for clustering, a distribution that later served as the foundation for deltaDEC. Additionally, it also offered a new perspective on the most effective strategies for addressing this task.

The success of the simpler distribution suggested that simpler models could work better for our data. Our scaled-down DEC models all include activation functions in the majority of their

hidden layers, making them *deep* models. One way to simplify these models would be to remove the activation functions, making the models *linear*. It is worth noting that our PCA reduction suggested potential linearity within the data, a trait that would be captured by these linear models, making this simplification a sensible step. One can see that without activation functions, our models become linear by following a data point pass through the whole network. In MA3K1, we learned that a neural network can be written as a weight matrix and a bias vector (dimension sizes can be found in the course notes). Therefore, travelling through the whole activation-function-free network can be thought of as a series of matrix-matrix and matrix-vector multiplications and additions. The final output for a data point x can then be written as $y = Wx + b$ where W and b are combinations of the previous layers' weights and biases. Further information on this can be found in MA3K1. The removal of activation functions has now made our models linear which will hopefully lead to better results for our data. We also simplified the autoencoder further by only using one hidden layer of size 10 (980 parameters in total - see MA3K1 for this calculation).

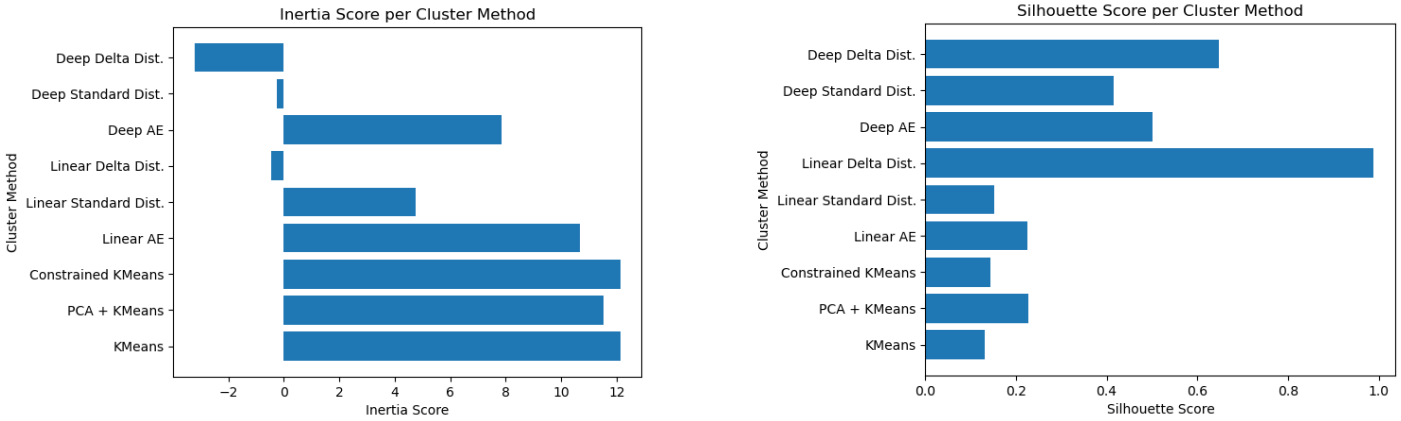
As with the deep models, we pre-trained and fine-tuned 5 separate models and took the median of results. We also only trained for 40 epochs due to the models being simpler and smaller (20 for the pre-training and fine-tuning each). The pre-trained linear autoencoder resulted in a significantly higher inertia compared with the deep version at 42894.258 (over a 1500% increase) as well as a much lower silhouette score of 0.225 (over a 50% decrease). This indicates substantially worse clustering but hopefully this will be resolved through fine-tuning. The standard DEC distribution linear model did reduce the inertia compared with the pre-trained linear model but did not reach the same low score as the deep standard distribution: attaining an inertia of only 117.024 (compared to 0.768). In addition, the silhouette score was also particularly poor at 0.1515, comparable with that of the KMeans clustering (0.131). The Delta distribution linear model, on the other hand, performed outstandingly well, potentially due to it being a combination of simplifying both the ideal probability distribution and the model itself. With an inertia of 0.625 and a silhouette score of 0.988, this approach led to near perfect clustering (recall perfect inertia and silhouette score are 0 and +1 respectively). We decided to name this method *deltaDEC* to fit with the naming convention in the field and also to reference the combination of elements from DEC and the Delta distribution. We appreciate that it is quite late into the essay to propose deltaDEC but we felt it was important to build up to the method and explain our reasoning behind its derivation.

5 Cluster Analysis

5.1 Cluster Algorithm Analysis

We have now discussed and tested numerous different clustering algorithms. Initially we started with KMeans to provide a baseline but found that this sometimes resulted in small, irrelevant clusters. To combat this, we then tried using a constrained KMeans with a minimum cluster size of 200 users which gave better results. One Hot Encoding the categorical columns of our data increased our data's dimensionality which threatened poor model performance. To mitigate this, we performed PCA reduction on our data to reduce the dimension size to 10 followed by KMeans, resulting in even better clustering results. Whilst the reduction indicated potential linearity in the data, the less than perfect explained variance suggested non-linear dimensionality reduction techniques could work even better. As a result, we explored deep learning's application to clustering, specifically implementing a scale-down version of the DEC algorithm. We analysed the performances of the pre-trained autoencoder as well as the standard distribution fine-tuned encoder but found fine-tuning with this distribution worsened the silhouette score. In response, we fine-tuned with a simpler distribution, the Delta distribution, leading to much improved clustering. Inspired by this, we aimed to simplify

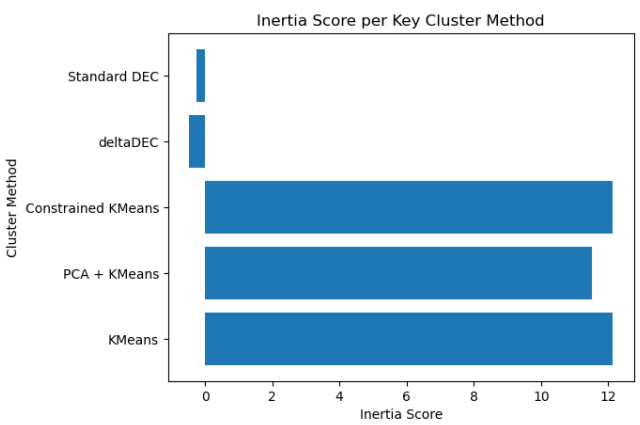
our models further by reducing the number of parameters and removing activation functions (i.e. making the models linear). Unfortunately, this led to significantly worse clustering for both the pre-trained linear autoencoder and the standard distribution fine-tuned linear model. That said, the Delta distribution fine-tuned linear model (or deltaDEC) produced near perfect clustering. We found that our two best performing models comprised the linear deltaDEC and the non-linear deep Delta Distribution. The explained variance from the PCA reduction would explain the success of both linear and non-linear techniques whilst the prominence of the Delta Distribution highlights that when working with small scale, real world data, using a simpler methodologies can work better. Below we provide a visual summary of this project's results. Note Figure 8 shows the inertia and silhouette scores of every model whilst Figure 9 displays only the most important. In order to visualise the wide range in inertia scores, we plot the log of the inertia which means certain models' inertias appear negative due to being less than 1. We also include a table (Table 1) where the most important models are in bold.



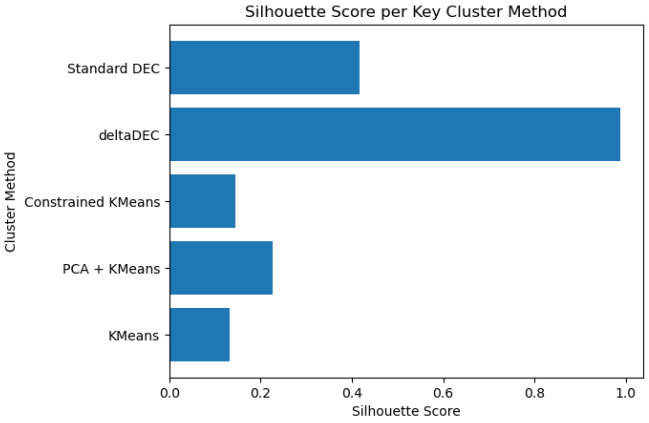
(a) Log Inertia Score per Algorithm

(b) Silhouette Score per Algorithm

Figure 8: All Models Analysis



(a) Log Inertia Score per Algorithm



(b) Silhouette Score per Algorithm

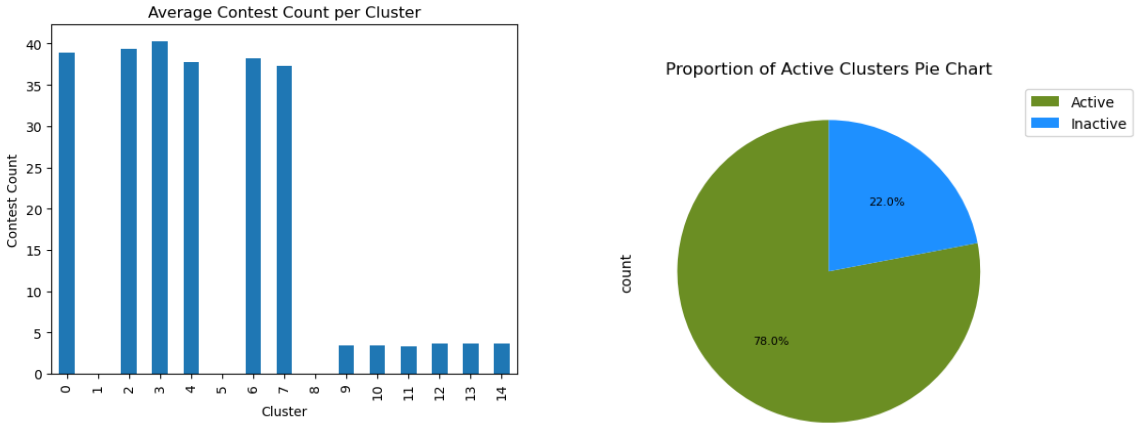
Figure 9: Important Models Analysis

Model	**Inertia**	**Silhouette Score**
KMeans	**186202.674**	**0.131**
Constrained KMeans	**185787.599**	**0.144**
PCA + KMeans	**101296.187**	**0.226**
Linear AutoEncoder	42894.258	0.225
Deep AutoEncoder	2605.904	0.501
Linear Standard Distribution	117.024	0.152
Standard DEC (Deep Standard Distribution)	**0.768**	**0.416**
deltaDEC	**0.625**	**0.988**
Deep Delta Distribution	0.040	0.648

Table 1: Cluster Metrics Comparison

5.2 Cluster Data Analysis

The performance of deltaDEC also extends to agreeing with our initial data analysis. We took the deltaDEC model with the highest silhouette score and passed our original data through this model to make cluster-specific dimensionally reduced representations. We then clustered these representations using Constrained KMeans and analysed the results. By observing the contest count (Figure 10a) per cluster, we identified Clusters 1, 5 and 8 as inactive accounts. These clusters made up 22% of users (Figure 10b), which fits exactly with our Exploratory Data Analysis (Figure 1c). This implies the clusters found by deltaDEC are meaningfully significant, giving confidence to conclusions drawn from them.



(a) Contest Count per Cluster

(b) Proportion of Active Clusters

Figure 10: Clustering Data Analysis

Further data analysis can be found in Appendix B where we explore the split in clusters seen in Figure 10a amongst other topics.

6 Conclusion

This research proposes deltaDEC, a simplification of the current SOTA clustering approach that can produce near perfect clustering for a fraction of the computational cost. This work could be of particular importance for SMEs looking to incorporate SOTA clustering into their analytics and business intelligence.

To illustrate the necessity for a SOTA clustering alternative for industry, the scaled down version of DEC required approximately 3.5 seconds to train one epoch on a *Central Processing Unit (CPU)*, a fundamental component of computer hardware found in every desktop and laptop. Achieving the SOTA level of training (half a million epochs) would have demanded over 20 days of continuous training. Moreover, this duration applies to the scaled-down version only, a model comprising just over 5500 parameters. The original model, however, containing over 2 million parameters, would have demanded an even more extensive training period making it entirely impractical for most businesses.

There are specialised computer hardware options specifically for deep learning tasks, such as a *Graphics Processing Unit (GPU)*, which can greatly accelerate training; however, buying or renting these can be costly for a company. Google Cloud [44] is an example of a GPU renting service which we explored for this project. Their cheapest GPU, an *NVIDIA T4*, charged \$0.35 (£0.28) per hour of training. On this GPU, the scaled down version of DEC took roughly 0.4 seconds to train one epoch which, at half a million epochs, would require just over 55 hours of training, costing just under \$20 (roughly £16). Whilst this could be an acceptable one-off investment, monthly or quarterly re-training to assess how trends evolve would become expensive, especially when compared with deltaDEC. As a free alternative, deltaDEC required just under 5 minutes to fully train on a CPU and resulted in near perfect clustering (Table 1), presenting a realistic alternative for companies looking for quick, high quality clustering at no financial cost.

6.1 Future Work

Given the evidence of linearity in our data (the explained variance in the PCA reduction - Figure 4a), it would be interesting to evaluate the performance of deltaDEC on non-linear datasets. Additionally, exploring how this approach compares to the original DEC on benchmark datasets like MNIST [1] would provide valuable insights. Furthermore, whilst deltaDEC can produce high quality clustering, on rare occasions it does not perform to this level. Although we have looked into why this occurs and how to stop it, no definitive answers have been found which could be an area for future research.

6.2 Summary

This essay discussed a range of clustering techniques and explored the theory behind fundamental concepts in deep learning and deep clustering. Our research showed that scaling down the SOTA approach does not retain its high clustering performance and that when working on a smaller scale, simpler models can work better. Finally, we proposed deltaDEC, a linear adaptation of the current SOTA approach that leverages the Nearest Cluster Delta Distribution, and showcased its remarkable ability to produce near perfect clustering for an incredibly low computational cost.

References

[1] J Xie et al. Unsupervised deep embedding for clustering analysis. *International Conference on Machine Learning*, 2016.

[2] What is machine learning (ml)? `https://www.ibm.com/topics/machine-learning`. Accessed: 2024-04-04.

[3] What is supervised learning? `https://www.ibm.com/topics/supervised-learning`. Accessed: 2024-04-04.

[4] R Adam et al. Deep learning applications to breast cancer detection by magnetic resonance imaging: a literature review. *BMC*, 2023.

[5] Q Rao et al. Deep learning for self-driving cars: Chances and challenges. *IEEE*, 2018.

[6] What is unsupervised learning? `https://www.ibm.com/topics/unsupervised-learning`. Accessed: 2024-04-04.

[7] S Shalev-Shwart et al. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[8] X Guo et al. Improved deep embedded clustering with local structure preservation. *International Joint Conference on Artificial Intelligence*, 2017.

[9] Z. Zheng et al. Deep clustering using 3d attention convolutional autoencoder for hyperspectral image analysis. *Nature*, 2024.

[10] S Shahriar et al. Let's have a chat! a conversation with chatgpt: Technology, applications, and limitations. *Artificial Intelligence and Applications*, 2023.

[11] Why one-hot encode data in machine learning? `https://machinelearningmastery.com/why-one-hot-encode-data-in-machinelearning/`. Accessed: 2024-04-08.

[12] Explaining sparse datasets with practical examples. `https://www.analyticsvidhya.com/blog/2022/11/explaining-sparse-datasets-with-practical-examples/`. Accessed: 2024-04-08.

[13] When and why to standardize your data. `https://builtin.com/data-science/when-and-why-standardize-your-data`. Accessed: 2024-04-08.

[14] What is feature scaling and why is it important. `https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/`. Accessed: 2024-03-07.

[15] Standardscaler vs. minmaxscaler vs. robustscaler: Which one to use for your next ml project? `https://medium.com/@onersarpnalcin/standardscaler-vs-minmaxscaler-vs-robustscaler-which-one-to-use-for-your-next-ml-project-ae5b` Accessed: 2024-02-29.

[16] sklearn.cluster.kmeans. `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`. Accessed: 2024-02-29.

[17] J Han et al. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2012.

[18] R Thorndike. Who belongs in the family? *Psychometrika*, 1953.

[19] E Schubert. Stop using the elbow criterion for k-means. *Association for Computing Machinery*, 2022.

[20] scikit-learn. `https://scikit-learn.org/stable/`. Accessed: 2024-04-09.

[21] P Bradley et al. Constrained k-means clustering. *Microsoft*, 2000.

[22] P Bradley et al. Clustering via concave minimization. *Conference on Neural Information Processing Systems*, 1997.

[23] Josh Levy-Kramer. k-means-constrained, April 2018.

[24] T Zhang et al. Big data dimension reduction using pca. *IEEE*, 2016.

[25] Principal component analysis: a review and recent developments. `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4792409/`. Accessed: 2024-04-24.

[26] D Bank et al. *Machine Learning for Data Science Handbook*. Springer Cham, 2021.

[27] A gentle introduction to dropout for regularizing deep neural networks. `https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/`. Accessed: 2024-04-13.

[28] What is a neural network? `https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/`. Accessed: 2024-04-12.

[29] Variational autoencoders are beautiful. `https://www.compthree.com/blog/autoencoder/`. Accessed: 2024-03-04.

[30] Y Chen et al. *AI Computing Systems*. Morgan Kaufmann, 2023.

[31] Tong Zhang Rie Johnson. Accelerating stochastic gradient descent using predictive variance reduction. *Advances in Neural Information Processing Systems 26 (NIPS 2013)*, 2013.

[32] D Kingma et al. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2015.

[33] Y Liu et al. An improved analysis of stochastic gradient descent with momentum. *Conference on Neural Information Processing Systems*, 2022.

[34] Andrew ng's deep learning course on deeplearningai. `https://www.youtube.com/watch?v=lAq96T8FkTw`. Accessed: 2024-02-29.

[35] G Garrigos et al. Handbook of convergence theorems for (stochastic) gradient methods. 2023.

[36] Sgd with momentum papers with code. `https://paperswithcode.com/method/sgd-with-momentum`. Accessed: 2024-02-29.

[37] Hyper parameter—momentum. `https://medium.com/ai%C2%B3-theory-practice-business/hyper-parameter-momentum-dc7a7336166e`. Accessed: 2024-03-04.

[38] Adam. https://optimization.cbe.cornell.edu/index.php?title=Adam. Accessed: 2024-04-09.

[39] Rmsprop. https://paperswithcode.com/method/rmsprop. Accessed: 2024-04-09.

[40] Pan Zhou et al. Towards theoretically understanding why sgd generalizes better than adam in deep learning. *NeurIPS*, 2020.

[41] Google's machine learning foundation course: Generalization. https://developers.google.com/machine-learning/crash-course/generalization/video-lecture. Accessed: 2024-04-12.

[42] Ma3k1. https://moodle.warwick.ac.uk/course/view.php?id=60821. Accessed: 2024-04-13.

[43] The dirac delta function. https://math.libretexts.org/Bookshelves/Differential_Equations/Introduction_to_Partial_Differential_Equations_(Herman)/09%3A_Transform_Techniques_in_Physics/9.04%3A_The_Dirac_Delta_Function. Accessed: 2024-04-13.

[44] Google cloud gpu pricings. https://cloud.google.com/compute/gpus-pricing. Accessed: 2024-04-24.

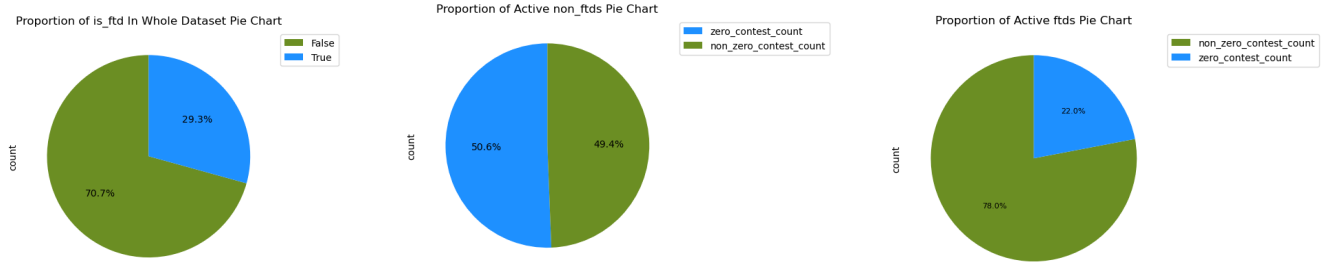
[45] Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *NeurIPS*, 2017.

A Exploratory Data Analysis Continued

To make this appendix fully self contained, the first paragraph of analysis is repeated from the essay. Beyond this, however, we push this analysis further.

Prior to employing any machine learning techniques, it's common practice to thoroughly investigate and analyse your dataset. This can help with initial pattern and anomaly detection as well as model selection.

A feature key to ParlayPlay's data was whether a user had deposited any money into their account (First Time Deposit - ftd) as this would indicate a commitment to staying and engaging with the platform. Analysis showed that over 70% of all accounts placed no deposit with the platform (Figure 11a) and of these, over 50% (Figure 11b) placed no contests - i.e. were completely inactive. On the other hand, of those who did place a deposit, only 22% did not actively engage with the app (Figure 11c) meaning, over the whole user base, just over 20% of users placed a deposit and were actively using the platform. These users are the most important from a business perspective and so we decided to perform the cluster analysis on these accounts.



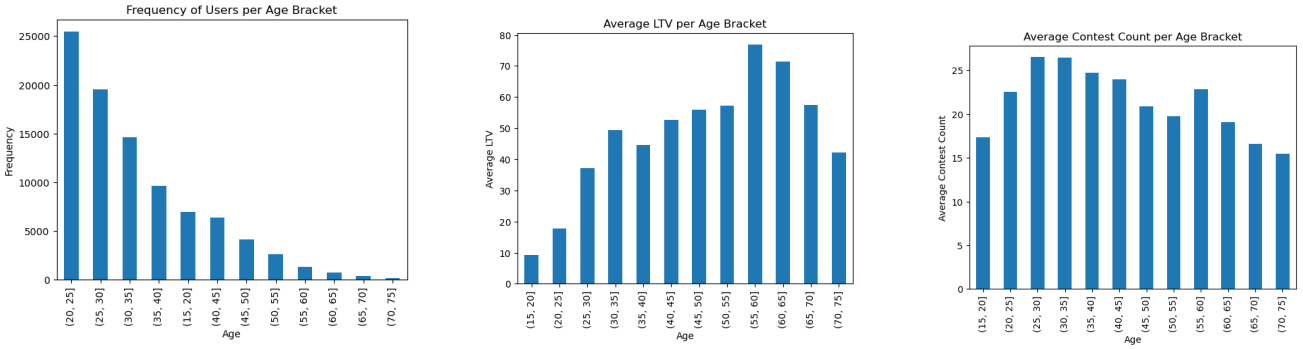
(a) Proportion of ftds in Whole Dataset

(b) Proportion of Active non_ftds

(c) Proportion of Active ftds

Figure 11: Analysis of ftd Users

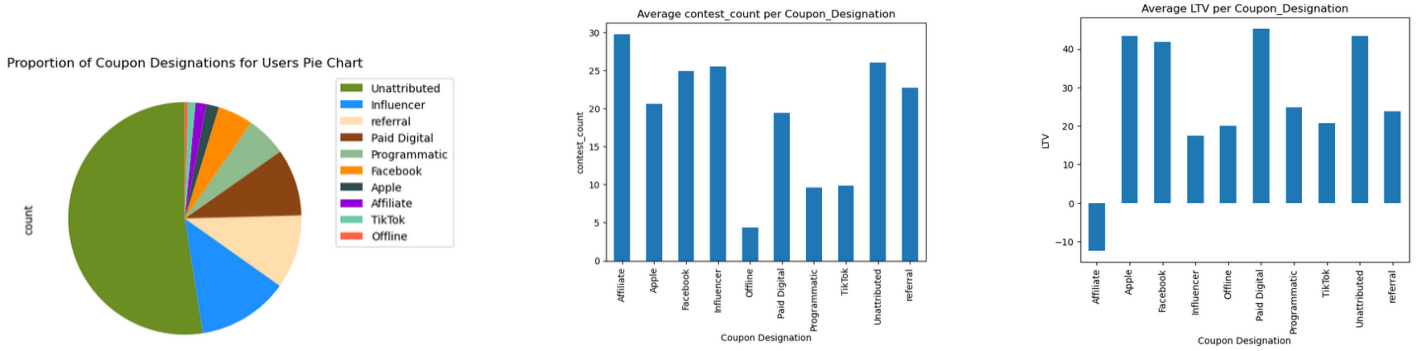
Given it would be the focus of our cluster analysis, we also chose to explore the ftd only data as well. We observed that although the majority of users are younger (Figure 12a), older users demonstrate higher engagement and profitability for the company (Figure 12b & 12c). As individuals age, their earning potential increases, leading to greater disposable income, which likely accounts for the trends seen in figures 12b and 12c. This is an encouraging sign for ParlayPlay, provided younger users remain loyal to the platform as they mature and their disposable incomes continue to grow.



(a) Frequency per Age (b) LTV per Age (c) Contest Count per Age

Figure 12: Analysis of Age

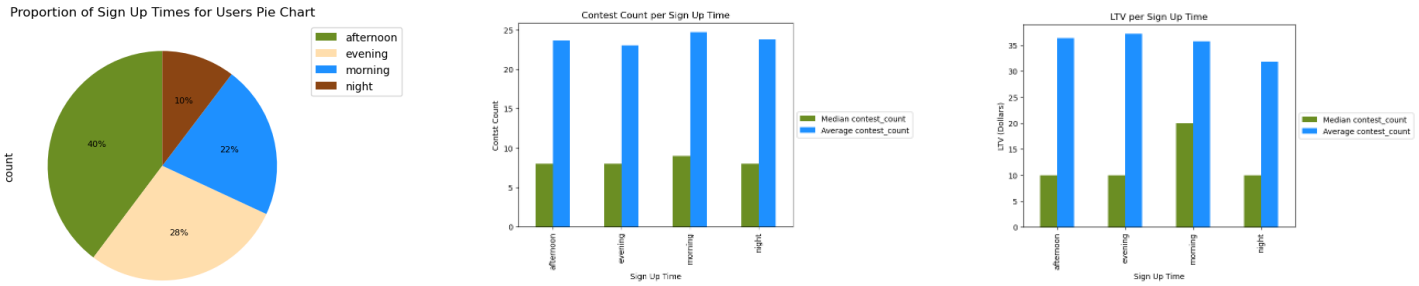
Additionally, we investigated how the user's chosen sign-up method (coupon designation) influenced their engagement and profitability for the company. We observed that the majority of users (over 50% - Figure 13a) did not use a coupon upon joining the platform. Nevertheless, this group displayed an average engagement (the moderate contest count in Figure 13b) and a high profitability (the high LTV in Figure 13c), which bodes well for the company given this group's size. Interestingly, the "Affiliate" coupon designation resulted in a negative LTV (Figure 13c) alongside a moderate to high contest count (Figure 13b). With close to 1400 users falling into this category, this raises significant concerns and could be an area for further study.



(a) Proportion of Coupon Designations (b) Contest Count per Coupon Designation (c) LTV per Coupon Designation

Figure 13: Analysis of Coupon Designation

Moreover, we also explored how the time a user joins the platform influences their engagement. We discovered that **Afternoon** was the most common time for users to sign-up (Figure 14a) and resulted in both average profitability and engagement (moderate LTV and contest count in Figures 14b and 14c). On the other hand, **Morning**, representing 22% of users, exhibited the highest LTV by a significant margin (Figure 14b) along with the highest contest count (Figure 14c). One potential explanation for this trend could be that signing up in the morning occurs outside the context of sporting events, potentially attracting more dedicated users, thus leading to higher contest count and LTV.



(a) Proportion of Sign Up Times (b) Contest Count per Sign up Time (c) LTV per Sign Up Time

Figure 14: Analysis of Sign Up Time

The influence of company policies is evident in the data. ParlayPlay identified that certain users were exploiting their system by selecting 5 or 6 picks per contest, leading to significant wins. These users were promptly identified and their accounts limited (restricted from further play). The shift in graphs at intervals [5.0, 5.5] and [5.5, 6] in Figure 15 illustrates the impact these users had on the company.

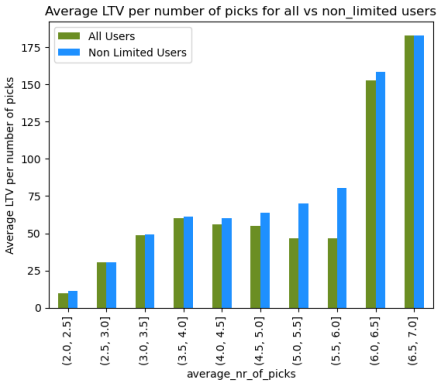
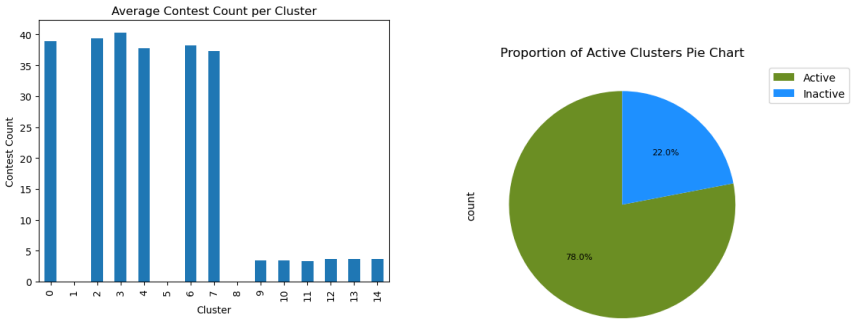


Figure 15: Impact of Limiting Certain Users

B Cluster Data Analysis Continued

As with Appendix A, the initial analysis here can also be found in the essay but beyond that, we discuss new topics.

The performance of deltaDEC also extends to agreeing with our initial data analysis. We took the deltaDEC model with the highest silhouette score and passed our original data through this model to make cluster-specific dimensionally reduced representations. We then clustered these representations using Constrained KMeans and analysed the results. By observing the contest count (Figure 16a) per cluster, we identified Clusters 1, 5 and 8 as inactive accounts. These clusters made up 22% of users (Figure 16b), which fits exactly with our Exploratory Data Analysis (Figure 13c). This implies the clusters found by deltaDEC are meaningfully significant, giving confidence to conclusions drawn from them.

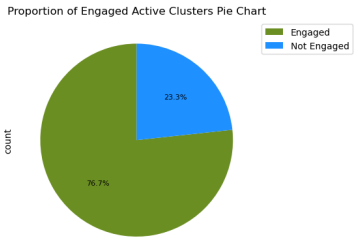


(a) Contest Count per Cluster

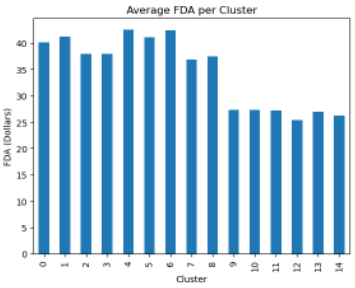
(b) Proportion of Active Clusters

Figure 16: Active vs Inactive Clusters

Figure 16a is particularly interesting as it suggests a clear dichotomy in the levels of engagement with the platform. Although users with poor engagement constitute just under 25% of active accounts (Figure 17a), their First Deposit Amounts (FDAs) are noticeably less on average than the engaged users, roughly 30% less (Figure 17b). This dichotomy extends through multiple features (average entry amount, average number of picks etc) and so there is lots of scope for early identification of these users allowing ParlayPlay to act quickly to boost engagement.



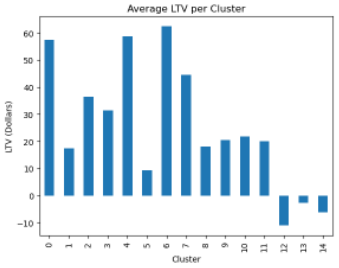
(a) Proportion of Engaged Active Users



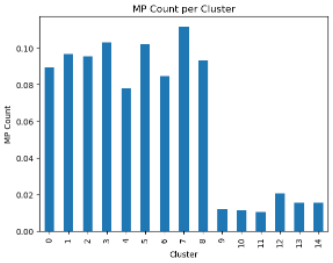
(b) FDA per Cluster

Figure 17: Exploring Split in Levels of Engagement of Active Users

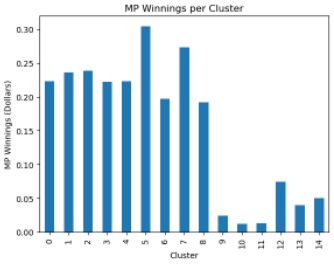
Slightly worrying, perhaps, are the three clusters with negative LTV (Figure 18a), which make up just over 4000 users (5% of total users). Analysing the MP Contests shows that of the disengaged active users, these clusters partake the most (Figure 18b) and win the most (Figure 18c). This could indicate a vulnerability with the system that is being exploited by these users, potentially warranting further exploration.



(a) LTV per Cluster



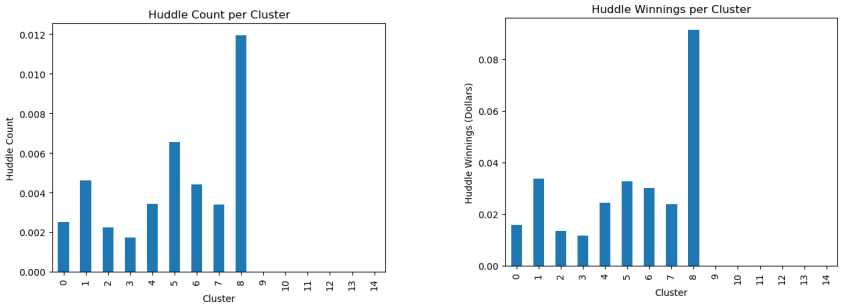
(b) MP Count per Cluster



(c) MP Winnings per Cluster

Figure 18: Exploring Negative LTV Clusters

It is also worth noting that the inactive Cluster 5 has the highest winnings from MP Contests. In addition to this, we also see that the inactive Cluster 8 has the highest Huddle count (Figure 19a) and winnings (Figure 19b) by a substantial margin. Their popularity amongst inactive users makes sense as users who do not engage financially with the platform would most likely look to partake in free contests. That said, ParlayPlay could capitalise on this popularity by enticing users to deposit money through a similar style paid contest. Given inactive users make up 22% of the total user base (Figure 11c) and nearly 10% of the total LTV (see code), this could be a substantial market to tap into.



(a) Huddle Count per Cluster (b) Huddle Winnings per Cluster

Figure 19: Exploring Huddle Contests

In addition to data analysis, understanding the features that make up each cluster can also provide interesting insights. To do so we analysed the features important in the actual deep learning model using *SHAP* values [45]. This technique is adopted from game theory and helps explain the workings inside machine and deep learning models. Essentially, one sees how the model performs on all the possible subsets of the features and compares these results to get an average importance for each feature. We found that the sign up time played an important role as well as the first SP result and contest count (Figure 20). Figure 18a would explain why the contest count could be an important feature but data analysis has not shown revealed any significant insights from the sign up time and first SP results. Furthermore, the importance of these two features does not really fit with ParlayPlay's current understanding of user's behaviour and as such, is an area for potential further analysis.

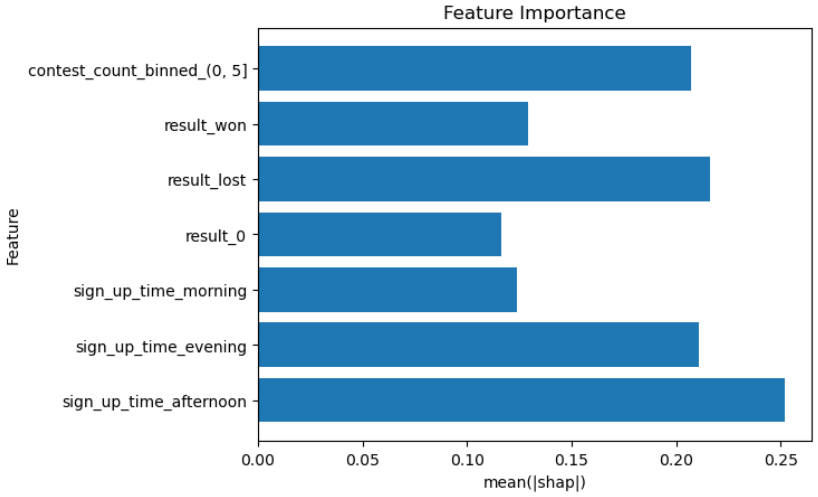


Figure 20: Top 8 Features

C Code

The code for all the data analysis and experiments can be found on my GitHub: `https://github.com/TomBidewell/Deep_Clustering_Project` .