LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK

FORSCHUNGSGRUPPE
DATA MINING IN DER MEDIZIN

Bachelor Thesis
in Computer Science

Attention in Mixed-Type Clustering

Jaanis Fehling

Aufgabensteller:	Prof. Dr. Christian Böhm
Betreuer:	Walid Durani
Abgabedatum:	tt.mm.yyyy

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.
This paper was not previously presented to another examination board and has not been published.

Munich, tt.mm.yyyy

.....
Jaanis Fehling

Abstract

This document serves as a model for the development of a thesis at the Department of Database Systems at the Institute for Computer Science at the LMU Munich. The abstract should not contain more than 300 words.

Contents

Chapter 1

Introduction

Clustering is an unsupervised machine learning method that groups similar observations together. Due to its ability to find patterns in an unlabeled dataset, its an essential task in Data Mining and Knowledge Discovery. A *cluster* is a group of similar instances that belongs to a *centroid* (center point of a cluster). Distance-based clustering algorithms use distance measures such as Euclidean distance to calculate the similarity of datapoints. Hierarchical methods partition the instances and merge (agglomerative) or split them into bigger or smaller clusters. Many other methods exist, but this work focuses on methods for clustering *mixed-type* data. [1]

Chapter 2

Related Work

Chapter 3

Foundation

3.1 Methodology

3.1.1 Datasets

In this work we use 8 mixed-type datasets from the UC Irvine Machine Learning Repository [18]. The Abalone dataset [21] contains physical measurements from abalones. It has 4177 instances, one categorial feature and seven continuous features. The Auction Verification dataset [22] has 2043 instances that contain verification runs of multi-round auctions. It is composed of six categorial and one continuous feauture. The Bank Marketing dataset [20] is related to a direct marketing campaign of a portuguese banking institution. It has 49732 instances, but was downsampled to 5000 random instances. The "age", "day" and "month" features were removed, which results in eight categorial and five continuous features. The Breast Cancer dataset [31] contains 699 instances and 9 categorial features. The Census Income dataset [14] has a total of 48842 instances. It was downsampled to 5000 random instances. It is composed of eight categorial and 6 continuous features. The Credit Approval dataset [26] contains information of applications for credit cards. It has 690 instances, nine categorial features and six continuous features. The Heart Disease dataset [13] is composed of four datasets and has 920 instances in total. It has seven categorial features and six continuous features. The Soybean (Large) Dataset [21] consists of 683 instances from soybeans with a certain disease and has 35 categorial features.

All instances containing missing values were removed. Duplicate instances were explicitly not removed, since there is no information available if they are duplicates by accident, or real duplicates. Categorial columns were standardized by removing the mean and scaling to unit variance, using the scikit-learn Python Library [24]. Formally, the standardized score z of a sample x from a

feature is calculated as

$$z = \frac{(x - \mu)}{\sigma}$$

where μ is the mean of the samples $x_1, ..., x_N$ from a feature of length N, defined as

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i$$

and σ is the standard deviation of the samples of a feature, defined as

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2}.$$

The datasets were shuffled. For all random operations, a random state of integer value 0 was used to ensure reproducibility. When using the PyTorch Python Libary [23] for implementing neural networks, a flag was set to only use deterministic algorithms.

3.1.2 Evaluation

The clustering results were each evaluated with two measurents, *Accuracy* and *Normalized Mutual Information*. Accuracy is defined as

$$\frac{\text{Instances predicted correctly}}{\text{Total number of instances}}.$$

Since any clustering algorithm will assign arbitrary class labels to each instance that might not allign with the ground truth class labels, we map the predicted label classes to ground truth label classes in a way maximize the number of correctly predicted instances. This is an inversion of the *Linear Assignment Problem*, we use an implementation provided by the Scipy Python Library [30] to solve this problem and find the optimal solution.

As shown in Chapter 5, some clustering methods falsely assign almost all instances to one target class. Because a part of the datasets we used are heavily imbalanced, this leads to unjustified high accuracy scores. Therefore, we use another metric, Normalized Mutual Information. It is based on *Mutual Information*, which, for the ground truth labels U and the predicted labels V (switching both variables will not change the outcome) is defined as

$$MI(U, V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} \frac{|U_i \cap V_j|}{N} log \frac{N|U_i \cap V_j|}{|U_i||V_j|}$$

where $|U_i|$ is the number of instances assigned to cluster U_i and $|V_i|$ is the number of instances assigned to cluster V_i [24]. Normalized Mutual Information is Mutual Information normalized by the mean of the Entropies of U and Y [24]:

$$NMI(U,V) = \frac{MI(U,V)}{\frac{1}{2}(H(U)H(V))}.$$

3.2 Classical Methods for clustering Mixed-Type data

3.2.1 k-means

The most well known distance-based clustering method is k-means [16]. The goal is defined as follows: Suppose we have a finite set of n instances $S = \{p_1, p_2, ..., p_n\} \in \mathbb{R}^m$ for a dataset with m features, the target of k-means is to find optimal centroids $B = \{b_1, b_2, ..., b_k\} \subseteq \mathbb{R}^m$ for a given $k(\leq n) \in \mathbb{N}$ that minimize the sum of the squared Euclidean distance of each point in S to its nearest centroid. Formally

$$\sum_{i=1}^{n} d(p_i, B)$$

has to be minimized, where d is the Euclidean distance from a point $p_i \in S$ to the nearest centroid in B [17]:

$$d(p_i, B) = min_{1 \leq j \leq k} d(p_i, b_j).$$

The Euclidean distance between two points p and q in an n-dimensional Euclidean space is defined as

$$d(p,q) = ||p-q|| = \sqrt{(p_1-q_1)^2 + (p_2-q_2)^2 + ... + (p_n-q_n)^2}.$$

Finding the optimal centroids is a NP-hard problem, even for $d = 2$, as shown by Mahajan et al. [17]. The most common algorithm used for the k-means problem is a iterative refinement technique proposed by Lloyd [15]. It is defined as follows

1. Randomly set k initial cluster centroids $b_1^{(1)}, ..., b_k^{(1)}$.
2. Assign each obseration p_i to the nearest centroid using squared Euclidean distance. This splits our instances into S into k sets $\{S_1^{(t)}, ..., S_k^{(t)}\}$.
3. Recalculate the optimal position of each centroid using the mean distance to each instance assigned to the centroid:

$$b_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{p_j \in S_i^{(t)}} p_j.$$

4. Repeat steps 2. and 3. until the centroid assignments no longer change.

3.2.2 k-modes

In many real-world scenarios, besides continuous, numerical data, *categorial* data exists. While Euclidean distance or other distance measures work well with continuous data, categorial data is different. Suppose we have categories $\{A, B, C\}$ of a given feature, we would encode them into numeric values to allow for computation of a distance measure:

$$\{A, B, C\} \equiv \{1, 2, 3\}.$$

While A and C can share the same semantic similarity as A and B, numerically category A and Category C are now $|1 - 3| = 2$ apart, while Category A and B are only $|1 - 2| = 1$ apart. During clustering, this could lead to instances being assigned to centroids based on a wrong distance assumption.

A possible solution is to use *one-hot encoding*, also known as *dummy coding* in classical statistics. One-hot encoding turns a discrete feature containing k mutually exclusive categories into a vector x of length k, in which only one of the elements x_k equals 1 and all remaining elements equal 0 [6]. For an instance B of a feature having $k = 3$ separate categories $\{A, B, C\}$, the one-hot vector x would be represented by $x = (0, 1, 0)^\intercal$.

According to Huang [11], one-hot encoding has two drawbacks:

1. In real-world applications, categorial features with hundreds or thousands of categories are encountered. This would result in a large number of binary features in the one-hot encoded representation, which will increase cost and space of computation.

2. The centroid value of a certain one-hot encoded feature, given by a real value between 0 and 1, cannot indicate the characteristics of the according cluster, since the feature only describes the presence or absence of one category.

Therefore, Huang [11] proposed using the Kronecker-Delta as a dissimilarity measure between multiple categorial features. Formally, if we have two instances X and Y of a dataset with m categorial features, d_1 will count the number of mismatches between the categorial features of both instances, defined as

$$d_1(X, Y) = \sum_{j=1}^{m} \delta(x_j, y_j)$$

where the Kronecker delta $\delta(x_j, y_j)$ is defined as

$$\delta(x_j, y_j) = \begin{cases} 0 & (x_j = y_j) \\ 1 & (x_j \neq y_j) \end{cases}.$$

If we have a finite set of n instances $S = \{p_1, p_2, ..., p_n\}$ for a dataset with m categorial features, the goal of *k-modes* [11] is to find optimal *modes* $B = \{b_1, b_2, ..., b_k\}$ for a given $k(\leq n) \in \mathbb{N}$ that minimize

$$\sum_{i=1}^{n} d_1(p_i, B)$$

where

$$d_1(p_i, B) = min_{1 \leq l \leq k} \sum_{j=1}^{m} \delta(p_{i,j}, b_{l,j}).$$

Similar to k-means, we can use an iterative algorithm for efficient computation [11]:

1. Randonly choose k instances from the dataset as initial modes for the clusters.
2. Assign each instance to their nearest mode using the proposed dissimilarity measure one by one and update the mode of each cluster after each assignment.
3. Test if each instance still belongs to its assigned mode, i.e. if each instance is assigned to its nearest mode. If the instance would belong to a different mode, reassign the instance and update the modes of both clusters.
4. Repeat step 3. until the mode assignments no longer change.

3.2.3 k-prototypes

As proposed by Huang [11], it is straightforward to combine the k-means and k-modes algorithms into the *k-prototypes* algorithm, which can be used to cluster *mixed-type* data (consisting of numerical, continuous and categorial features). The dissimilarity between two instances X and Y with features $A_1^r, A_2^r, ..., A_s^r, A_{s+1}^c, ..., A_m^c$, where features $A_1^r, ..., A_s^r$ are continuous and features $A_{s+1}^c, ..., A_m^c$ are categorial, is defined as

$$d_2(X, Y) = \sum_{j=1}^{s} (x_j - y_j)^2 + \gamma \sum_{j=s+1}^{m} \delta(x_j, y_j).$$

The first part of the equation is the Euclidean distance as used in k-means, while the second part is taken from the k-modes algorithm. Huang [11] states: "The weight γ is used to avoid favouring either type of attribute".

Again, we need to find k optimal centroids $B = \{b_1, b_2, ..., b_k\}$ and therefore have to minimize

$$\sum_{i=1}^{n} d_2(p_i, B)$$

where

$$d_2(p_i, B) = min_{1 \leq l \leq k} \sum_{j=1}^{s} (p_{i,j} - b_{l,j})^2 + \gamma \sum_{j=s+1}^{m} \delta(p_{i,j}, b_{l,j}).$$

We can minimize both distance measures at the same time since they are nonnegative. Therefore, we can use the same algorithm as defined in 3.2.2. [11]

3.2.4 Gower distance

Gower distance [9] is a general similarity measurement between instances containing mixed-type features. It is defined as follows: When comparing instances x_i and x_j, for each feature k of p total features, we calculate a score $s_{ijk} \in [0, 1]$. The score will be close to 1 for two instances x_{ik} and x_{jk} of a feature k if they are similar, and close to 0 they are not similar. Gower distance is also computable between instances with missing values, therefore a quantity δ_{ijk} is calculated, which is equal to 1, when feature k can be compared across the two instances x_i and x_j, and 0 otherwise (illustrated in Figure 3.1). Gower distance then is the average of the known score

$$S_{ij} = \frac{\sum_{k=1}^{p} s_{ijk}\delta_{ijk}}{\sum_{k=1}^{p} \delta_{ijk}}.$$

The Score s_{ijk} is calculated differently according to the type of feature [9]:

1. For *dichotomous* (when a value is either present or absent) features, the score s_{ijk} is 1 when the value is present in both features and 0 otherwise, as shown in Figure 3.1.

2. For categorial features, the score s_{ijk} is 1 if they both instances match on feature k and 0 otherwise.

3. For continuous features the score is calculated as

$$s_{ijk} = 1 - \frac{|x_i - x_j|}{R_k}$$

where R_k is the range of feature k in the dataset or in the sample.

SCORES AND VALIDITY OF DICHOTOMOUS CHARACTER COMPARISONS

	Values of character k			
Individual i	+	+	−	−
j	+	−	+	−
s_{ijk}	1	0	0	0
δ_{ijk}	1	1	1	0

Figure 3.1: Illustration from the original paper by Gower [9]. Score s_{ijk} and quantity δ_{ijk} of a feature k on two instances x_i and x_j. Presence of a feature is denoted by "+" and absence by "-".

Gower [9] has shown that $\sqrt{1 - S_{ij}}$ is a valid distance representation for two instances x_i and x_j. We can now convert our similarity matrix S into a distance matrix and are able to use Hierarchical clustering methods [12]. Philip and Ottaway [25] used Gower distance with agglomerative clustering. Agglomerative clustering places each instance into its own cluster and recursively merges the clusters together using the given distance matrix, until only the specified number of clusters is remaining [12].

3.3 Deep Clustering

3.3.1 Neural Networks

The idea of computation by neurons inspired by the human brain was first formalized into a mathematical model by McCulloch [19]. A *neuron* was defined as a element that takes multiple boolean inputs and has one boolean output. The neuron *fires*, meaning the output is set to true, when the sum of the input values extends a certain threshold.

The single-layer *perceptron*, the first neural machine learning algorithm, was invented by Rosenblatt in 1957 [27]. Formally, the binary valued output o_j given an input vector x_i is calculated as

$$o_j = \begin{cases} 1 & \sum_i w_{ij} x_i + b > 0 \\ 0 & otherwise \end{cases}$$

where w is the learnable weight matrix, and b is a predefined bias. For training, the weights w are simply incremented when the output is 0 but the ground truth is 1, and decremented if the output is 1 but should be 0. If the output was predicted right, no weights are changed.

The idea of neural networks was revived three decades later, using multiple perceptron layers and a differentiable error function [28]. In a multi-layer neural network, we have an input layer, an output layer and multiple hidden layers. Each layer is a collection of neurons, that acquire the outputs of each neuron from the previous layer (or from the input in case of the input layer) as their input, and produce a new output. Formally, the output a_j of the jth neuron of layer n that gets d input values from the previous layer is defined as

$$a_j = \sum_{i=1}^{d} w_{ji}^{(n)} x_i + b_j^{(n)}$$

where $w_{ji}^{(n)}$ is the learnable weight of the input x_i going through neuron j in layer n [5]. A bias b_j is also added. This output is commonly referred to as an *activation* of a neuron [5]. Because every neuron is a linear function, in order to avoid the collapse of each neuron from all layers into a single linear function, we pass the activation a_j into a non-linear activation function f

$$z_j = f(a_j)$$

before being passed to the next layer of neurons [8].

As shown in a recent survey [8], there are many different activation functions used in neural networks. A simple example for an activation function, which naturally aligns with the idea of a biological neuron firing is the *step function* (also known as *Heaviside step function*) [5], that outputs 0 for negative values and 1 for positive values:

$$\text{step function}(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}.$$

In order to stay between 0 and 1, but also utilize all real values in between, another activation function that has historically been popular is the *logistic sigmoid* function [8], that squashes any input in $]0, 1[$:

$$\text{logistic sigmoid}(x) = \frac{1}{1 + e^{-x}}.$$

Because the ouputs are in a range close to 0, it is prone to the *vanishing gradient problem*. In the vanishing gradient problem, a gradient that is very

close to 0 leads to almost no update in the weights of the network during training. Moreover, using a exponential value naturally leads to a greater computational complexity. [8]

The current state-of-the-art activation function, the *Rectified Linear Unit* (ReLU) [8], is not limited by the above disadvantages. It is simple and computationally performant. It is defined as the identity for positive values and as 0 for negative values:

$$\text{ReLU}(x) = max(0, x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}.$$

One downside of the ReLU activation function is that there will be a substantial amount of dead neurons in the network (neurons with output 0 will not affect the neurons of the next layer). The *Leaky Rectified Linear Unit* (Leaky ReLU or LReLU) [8] utilizes negative values as well, but with a small coefficient, usually set to 0.01. It is defined as

$$\text{Leaky ReLU}(x) = \begin{cases} x & x \geq 0 \\ 0.01 \times x & x < 0 \end{cases}.$$

All four activation functions are illustrated in Figure 3.2.

In a multi layer neural network, we need to be able to update the weights during training, so the network can learn the most optimal weights in order to fulfill its designated task. In the single layer perceptron, we were able to change the weights directly based on the predicted output, but in a multi layer network this is non-trivial, since we have multiple such perceptron layers that are sequentially attached. The solution is to use differentiable activation functions (or activation functions differentiable at almost every point, e.g. ReLU is not differentiable at 0), so each neuron together with its activation function becomes a differential function of the input, the weight and bias. We can now define a differentiable error function of the network outputs, which therefore is also a differentiable function of the weights. A common error function used in neural networks is the *mean squared error*, defined as

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - z_i)^2$$

where n is the number of input instances, y_i ist the target label of instance x_i and z_i is the network output on instance x_i. The derivatives of the error function in combination with gradient descent can be then be used to find the weights that minimize the error function. [5]

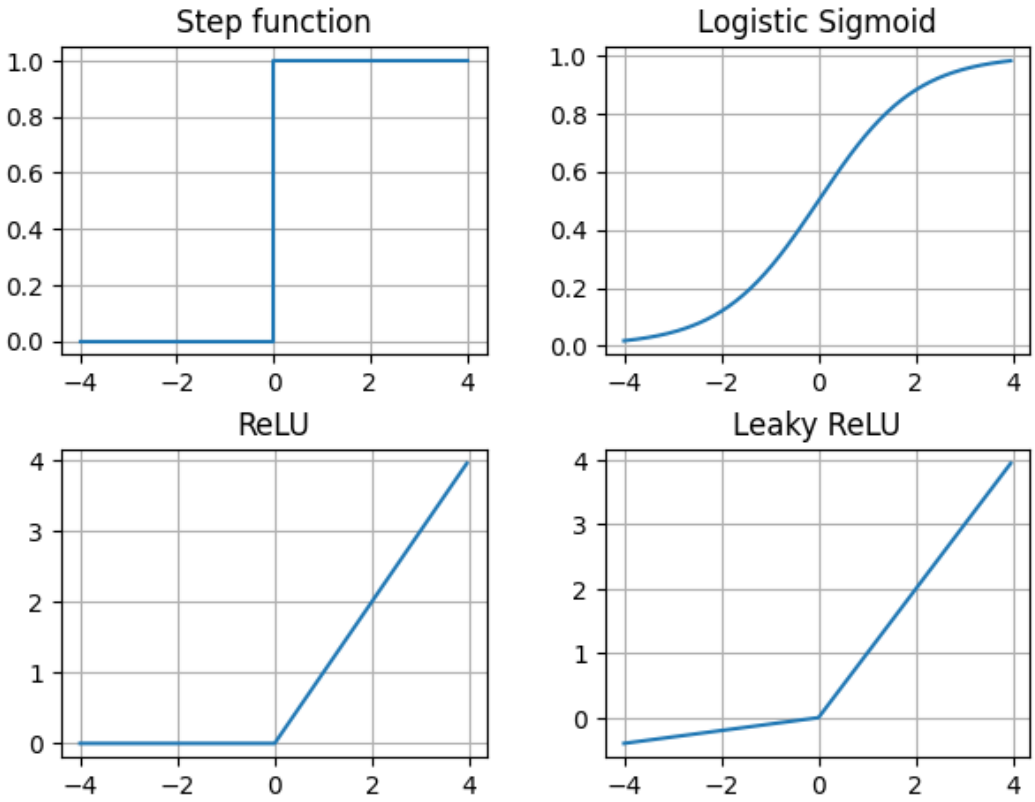

Figure 3.2: An illustration of the step function, the logistic sigmoid, the ReLU and the Leaky ReLU activation functions.

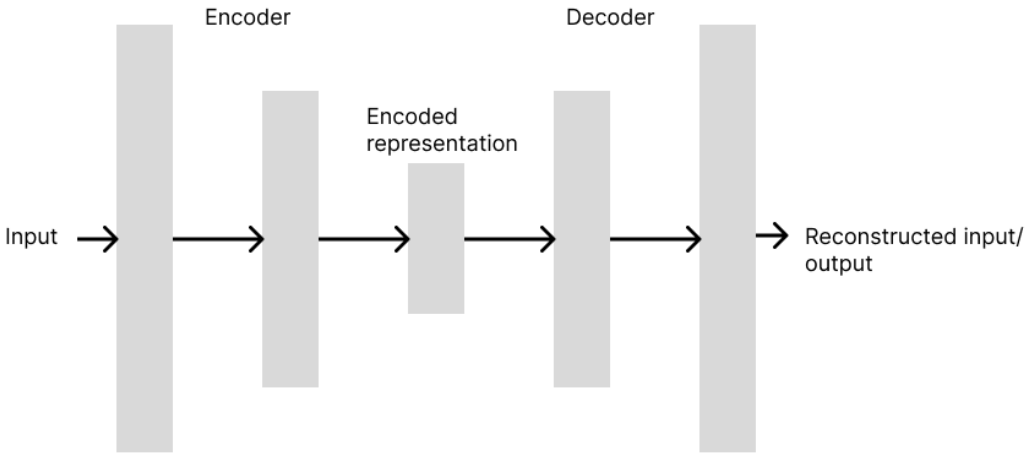

Figure 3.3: Illustration of an autoencoder neural network.

3.3.2 Autoencoder

A neural network, as defined in the previous section, needs the target labels of a dataset in order be trainable. In a unsupervised learning setting, this is unapplicable. An approach used for pre-training neural networks with unlabeled data is using an *autoencoder* neural network, first formalized by Ballard [4]. An autoencoder is a neural network, that projects the input into a lower-dimensional embedding space, and then back into its original dimensionality. The network tries to reconstruct the original input the best it can, after it was encoded into the embedding space. Therefore the target label fed into the loss function is the original input. This allows such a network to be trained without the need for labeled data. The autoencoder consists of two blocks, where each is a collection of fully connected linear layers. The *encoder* maps the input into the embedding space, while the *decoder* projects the encoded representation into its original dimensionality. Both blocks can contain multiple hidden layers, but typically they are constructed symmetrical, meaning the number and size of the linear layers in the decoder are the same as in the encoder, but inverted.

In unsupervised learning such as clustering, the autoencoder can be used to learn a dense representation of the input data. After training, we can feed the data into the encoder part of the autoencoder in order to obtain the encoded data, before utilizing a classical clustering algorithm to acquire the

final cluster labels. The encoded representation has a smaller dimensionality and only contains the most relevant parts of the data (which were needed for reconstruction), which can benefit clustering.

Chapter 4

Attention in Mixed-Type Clustering

4.1 Autoencoder and k-means

It is difficult to formulate statistical distance measures for comparing two values of a categorial feature. Categorial features are irregular in nature, they can be binary, ordinal, non-ordinal or composed of many semantically different classes. This makes it hard to cluster a mixed-type datasets with classical clustering methods. As described by Huang [11] and also explained in Section 3.2.2, one-hot encoding is suboptimal for the use in clustering tasks. The proposed solution by Huang, k-prototypes [11], does not beat k-means or k-means with one-hot encoding on any tested dataset, as shown in Chapter 5. Gower distance shows great accuracy results on all datasets and manages to beat the other methods on five out of eight datasets. As further illustrated in Chapter 5, the accuracy evaluation metric is highly problematic, since some datasets are heavily imbalanced. In fact, the datasets on which gower distance achieves the best accuracy results are greatly imbalanced. This means the agglomerative clustering used with gower distance falsely assigns almost all instances to one target class. When looking at Normalized Mutual Information (NMI), gower distance with agglomerative clustering does not beat k-means or k-means with one-hot encoding on any dataset. This leaves us with naive k-means intended for only numerical, continuous data and k-means with one-hot encoding, which is a inefficient, suboptimal encoding.

Therefore, instead of using statistical distance measure to compare instances with categorial and continuous features, we use an autoencoder neural network to learn a dense representation of the data. The clustering architecture we build upon is a multi-stage process. We first train the autoencoder on our dataset independently of the clustering process. The training step is repeating

many times, in our experiments we train the network a hundred times (hundred epochs). After the initial training step, we can feed the dataset into the encoder part of our trained autoencoder to get the dense, encoded representation of our data. We can perform k-means clustering over this representation to get the final cluster labels.

4.2 Column Embeddings

Even when using neural networks, there is still the problem of representing categorial values, especially non-ordinal values, as explained in depth in Section 3.2.2. Therefore, we use a learnable *embedding* layer for categorial values. The resulting embedded tensor of the embedding layer for a categorial feature is a meaningful representation that is more dense that a one-hot encoded feature. The embedding layer is composed of a weight tensor e_i for each categorial feature i. The weight tensor e_i has the following dimensionality: $|i| \times s_i$, where $|i|$ is the number of unique classes in categorial feature i and s_i is the corresponding embedding dimension. We calculate the embedding dimension s_i of feature i as

$$s_i = min(50, \lceil \frac{|i|}{2} \rceil).$$

We then encode the classes of each categorial feature with integer values, going from 0 onwards. These integer values serve as indices for the corresponding weight tensor of our embedding layer. Passing a value j of a feature i in the embedding layer returns the one-dimensional tensor at index j of the weight tensor e_j, which is of dimension $1 \times s_i$. The resulting tensors of each categorial feature from the embedding layer are concatenated into a tensor of size $1 \times \sum_{i=1}^{n} s_i$, where n is the number of categorial features. To construct the complete input tensor, we concatenate the tensor of the embedded categorial features with the tensor of the continuous features. The continuous features of an instance are not embedded, but scaled by removing the mean and scaling to unit variance as explained in Section 3.1.1. Having m continuous features would result in a tensor of continuous features with size $1 \times m$. After concatenating both tensors, this leaves us with an input tensor of size $1 \times (m + \sum_{i=1}^{n} s_i)$. This input tensor can then be passed into the first linear layer of the encoder.

The goal of the autoencoder therefore is to reconstruct the embeddings, not the actual input. To be able to use a loss function on this network, we clone the tensor of the embedded categorial features and save it in a variable, and later compare the decoder output with the cloned embedded tensor (concatenated with the continuous features).

4.3 Attention

Self-attention is a machine learning mechanism for sequence processing, originally used in machine translation by Bahdanau et al. [3] and extended into the *Transformer* architecture by Vaswani et al. [29]. It is a sequence-to-sequence operation, that aims to create a better representation of the input sequence by putting each element into the context of the other elements (paying "attention" to each other element). We do not operate with a sequence, but instead with multiple categorial features that might be related. For example, the relevance of a feature for the clustering process might depend of the value of another feature. The goal of the attention mechanism is to augment this feature (increase or decrease its vector representation) only when the other feature has this certain value.

Self-attention operates with three input tensors: *queries*, *keys* and *values*. Each of them is the input tensor, but used for different parts of the attention process. We add three independent linear projection to the input tensor to learn different representations for queries, keys and values. We apply attention to the categorial features after they were embedded in the embedding layer and then projected into queries, keys and values. The resulting vectors for each categorial feature are called *contextual embeddings*, since they contain information on the context of the input sample. We concatenate these embedding vectors lengthwise, before passing them into the first linear layer of the encoder, as illustrated in Figure 4.1.

Queries, keys and values are a sequence of n vectors that have d dimensions, resulting in dimensionality $n \times d$. In our case, the input dimension and therefore also the queries Q, keys K and values V are one-dimensional vectors, since the embedded vectors of each categorial features are concatenated along the first axis, as further illustrated in the previous section. We therefore cannot use our previously created input tensor. Because we calculate an individual embedding dimension (with a formula defined in the previous section) for each categorial feature, we cannot concatenate the vectors row-wise, since they all differ in length. Instead, we have to use a fixed embedding dimension d for every categorial feature. Then, we can create tensors Q, K and V with dimension $n \times d$ each, where n is the number of categorial features. The value 32 was chosen for embedding dimension d.

Because Attention is a mechanism that calculates values that combine multiple features, we need to be able to differentiate between the classes of different features, since we have a separate learnable embedding for each feature. A solution in the domain of text translation proposed by Vaswani et al. [29] is to add a positional encoding to each embedding in order to indicate the position of the word in the sentence. This is not suitable for tabular data, since the

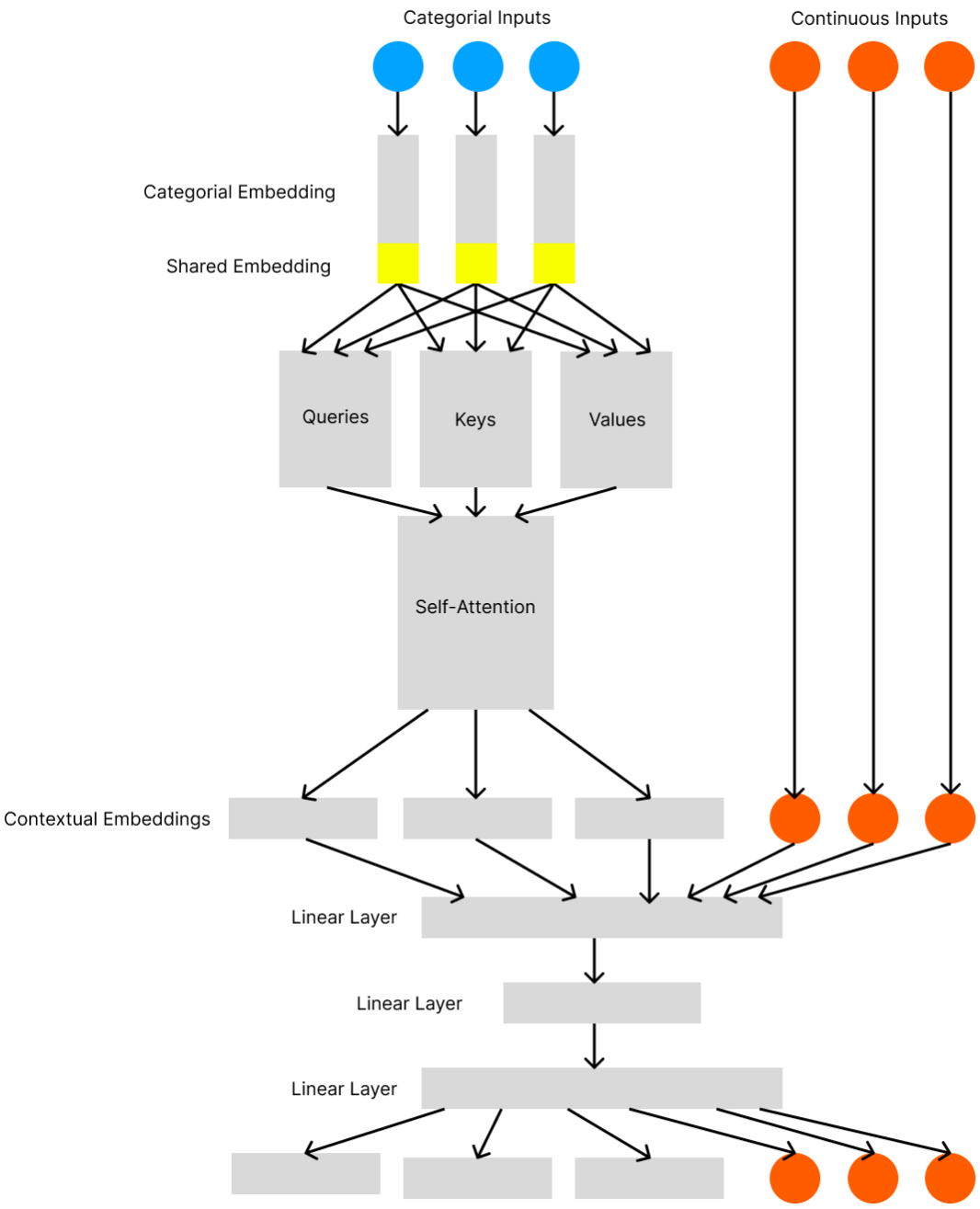

Figure 4.1: Architecture of the self-attention autoencoder network.

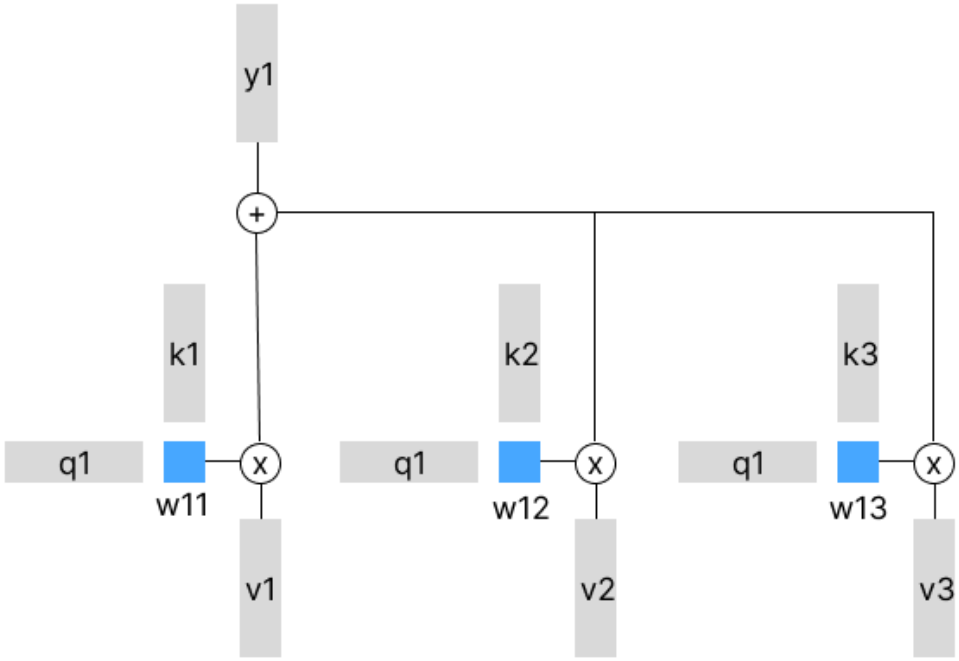

Figure 4.2: Illustration of the attention mechanism on calculating the first output vector, inspired by Bloem [7]. q1, k1, v1 represent the vectors at index 1 of Queries Q, Keys K and Values V respectively. The softmax function over the weights is not part of this illustration.

features do not have a particular order. We instead use a solution proposed by Huang et al. [10], which uses a shared embedding across all categorial features. The goal of the shared embedding is to learn dissimilar representations for different classes of separate features in order to distinguish them. The shared embedding is a learnable tensor of size $n \times \frac{d}{8}$, that is concatenated to our input tensor in the first dimension. The fixed embedding dimension d is therefore subtracted by 8, which results in a input tensor of categorial features of size $n \times d$.

The first step of self-attention is to calculate attention weights for every element of the input sequence. For that, we will multiply every query of our queries with the keys. Formally, we can define this as a matrix multiplication:

$$W = QK^\intercal,$$

where W are the resulting attention weights, Q are the queries and K the keys. The goal of this operation is to calculate weights W_{ij} that indicate how important feature i is to feature j and vice versa. Each weight is an arbitrary value, therefore we use the softmax function [5] to flatten the weights. The softmax function on a weight w_{ij} is defined as

$$\text{softmax}(w_{ij}) = \frac{\exp(w_{ij})}{\sum_j w_{ij}}.$$

Because the weights W can grow towards positive and negative infinity, in order to minimize the vanishing gradient problem of the softmax function (further explained in Section 3.3.1 in the case of the sigmoid function), we scale the attention weights by $\sqrt{d}$ before passing them into the softmax function:

$$W = \frac{QK^\intercal}{\sqrt{d}}.$$

We are scaling by $\sqrt{d}$, since its the average increase in Euclidean length of a weight vector w when increasing its dimension d. [29]

Now after we have obtained the attention weights that indicate how important a feature is to every other feature, we need to multiply these weights with each feature vector. This will result in vectors that carry information to which other feature vectors we need to pay attention to. We use the remaining linear projected tensor, the values V, for this step. The full Attention mechanism is therefore defined as

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^\intercal}{\sqrt{d}})V.$$

An illustration on how the first output vector in the attention mechanism is calculated is given by Figure 4.2.

4.4 Transformer

The *Transformer* is a architecture based on the attention mechanism, first formalized by Vaswani et al. [29]. The Transformer extends attention into *multi-head attention* and combines it with a feed-forward network and layer normalization. Multi-head attention [29] splits the input in h parts and learns h different representations for queries Q, keys K and values V respectively. Attention is then performed h times in parallel using the different representations of Q, K and V. This allows the model to perform attention over different subspaces of the embedded features at the same time. Using normal attention, we would only be able to attend to the complete embedding of the other features.

After performing attention, the Transformer adds the attention outputs to the initial input and normalizes this sum using *layer normalization* [2]. The output is then fed into two linear layers, the first layer projecting the input to four times its size, and the second layer projecting it onto the original size again. Once again, the output is added to the input of the linear layers, and normalized using layer normalization. This process, illustrated in Figure 4.3, is sequentially applied N times. The layer normalization and feed-forward network between each multi-head attention operation allow the model to parameterize the attention outputs. This makes sure the model can learn from each attention step.

The Transformer was originally designed for natural language tasks, but Huang et al. [10] proposed the *Tab Transformer*, a multi-layer perceptron classifier for the tabular domain, based on the Transformer architecture. Inspired by the performance increase on mixed-type datasets using this architecture, we build a Transformer-based autoencoder network for clustering. The architecture, illustrated in Figure 4.3, is similar to the previous attention-based autoencoder but uses $N = 6$ Transformer blocks instead of one single self-attention block. Our categorial feature embedding size is 32 as previously. We perform multi-head attention with $h = 8$ heads, yielding a embedding dimension for each head of $\frac{d}{h} = 4$.

4.4.1 FT-Transformer

Gorishniy et al. extends the Tab Transformer into the *FT-Transformer*, which uses embeddings for the numerical features as well. We implement a FT-Transformer autoencoder network, illustrated in Figure 4.4. Our implementation uses separate linear layers for each continuous column, that transform a singular value of a continuous feature into a $1 \times d$ vector representation, where d is the embedding dimension of the categorial features. This allows the con-

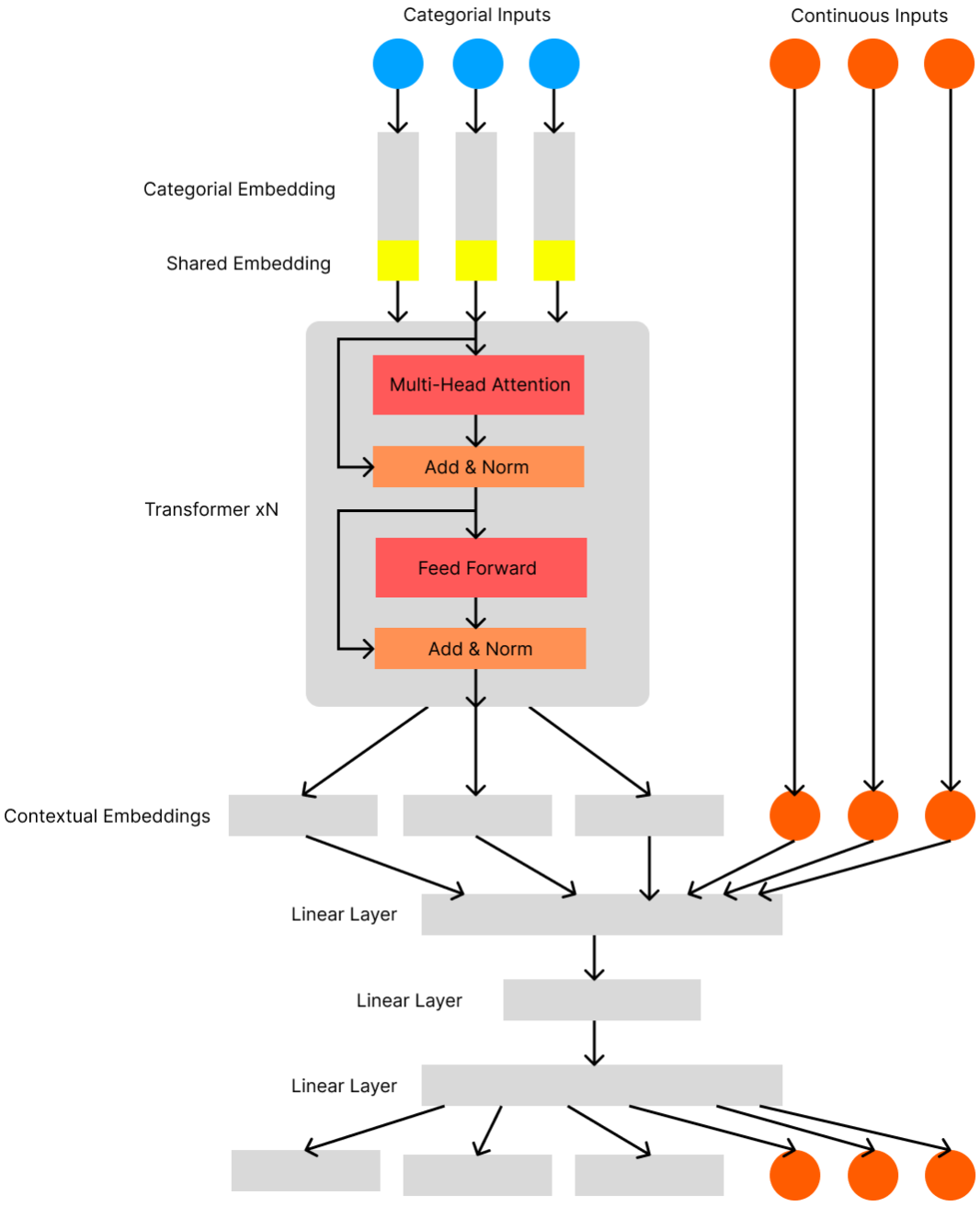

Figure 4.3: Architecture of the Transformer autoencoder network.

tinuous embeddings to be concatenated with the categorial embeddings, before being passed into the Transformer block. The rest of the implementation remains canonical to the previous architecture, only we do not concatenate the continuous feature values with the Transformer output, since the continuous features are already part of the Transformer block. The FT-Transformer paper does not use a shared embedding, we decided to keep still implement it, since Huang et al. [10] showed great accuracy gains when using a shared embedding.

The intuition behind using the FT-Transformer is that many continuous features in the tabular domain are only composed of a small range of values, and can therefore easily represented by an embedding vector. Using Attention over both continuous and categorial features allows the model to learn dependencies across both types of features.

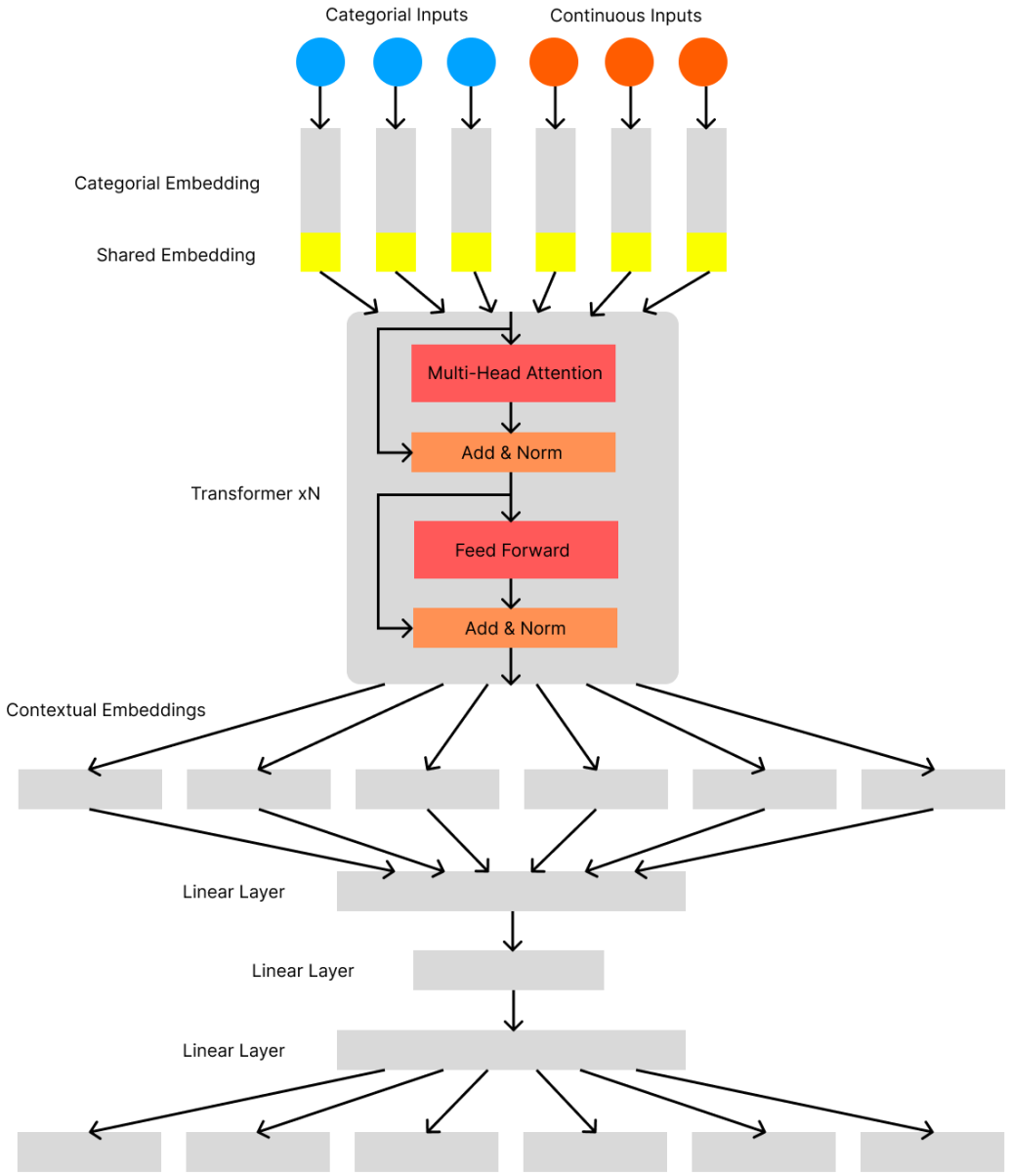

Figure 4.4: Architecture of the FT-Transformer autoencoder network.

Chapter 5

Experiments

5.1 Comparison of classical Clustering Methods

Our first comparison is between classical clustering methods for clustering mixed-type data. We evaluated k-means, k-means with one-hot encoding, k-prototypes and gower distance with agglomerative clustering, using average linkage (average distance between each instance of two sets).

	k-means	k-means one-hot	k-prototypes	Gower distance
Abalone	0.1718	**0.1740**	0.1716	0.1614
Auction Verification	**0.0162**	0.0071	0.0077	0.0062
Bank Marketing	0.0198	**0.0261**	0.0195	0.0013
Breast Cancer	**0.7468**	0.7363	0.5925	0.5537
Census Income	0.1080	**0.1850**	0.1417	0.0043
Credit Approval	**0.3131**	0.1710	0.1166	0.0035
Heart Disease	**0.2046**	0.1645	0.1893	0.1408
Soybean Disease	0.6722	**0.7102**	0.5676	0.6695

Figure 5.1: Comparison of Normalized Mutual Information of various classical methods on clustering mixed-type datasets.

	k-means	k-means one-hot	k-prototypes	Gower distance
Abalone	0.1353	0.1314	0.1343	**0.1954**
Auction Verification	0.6647	0.5761	0.5805	**0.8008**
Bank Marketing	0.7796	0.7866	0.7872	**0.8842**
Breast Cancer	**0.9605**	0.9502	0.9151	0.9004
Census Income	0.6082	0.6976	0.6256	**0.7684**
Credit Approval	**0.8086**	0.7060	0.6662	0.5482
Heart Disease	0.3344	0.3211	0.4247	**0.5652**
Soybean Disease	0.5765	**0.5996**	0.4715	0.501

Figure 5.2: Comparison of Accuracy of various classical methods on clustering mixed-type datasets.

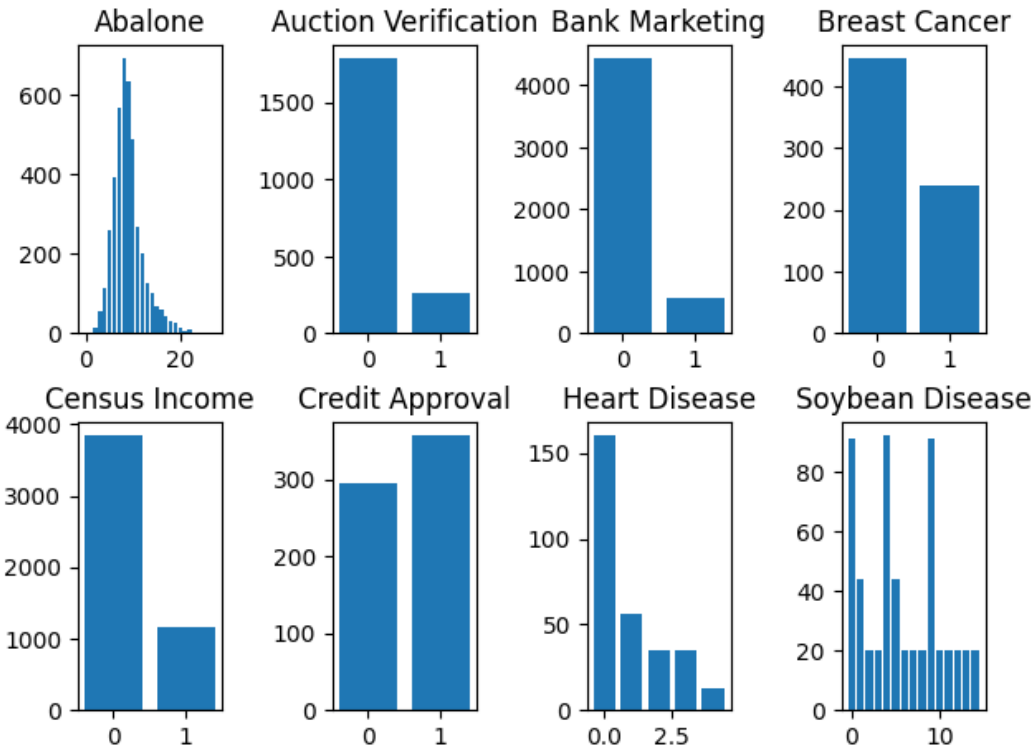

Figure 5.3: Number of instances of each target class of each dataset used.

	No Column Embedding AE	All Columns AE	Categorial Columns AE	All Columns to Categroial Columns AE
Abalone	0.1559	0.1695	0.1684	**0.1709**
Auction Verification	0.0066	0.0012	0.0002	**0.0413**
Bank Marketing	0.0015	0.0039	**0.0197**	0.0000
Breast Cancer	**0.7602**	0.7178	0.2878	0.6927
Census Income	0.0003	0.0241	**0.1024**	0.0042
Credit Approval	0.0031	**0.0097**	0.0008	0.0028
Heart Disease	0.1389	0.1663	**0.1848**	0.0947
Soybean Disease	0.4974	0.4836	0.4169	**0.5426**

Figure 5.4: Comparison of Normalized Mutual Information of various Autoencoder architectures combined with k-means on clustering mixed-type datasets.

	No Column Embedding AE	All Columns AE	Categorical Columns AE	All Columns to Categorical Columns AE
Abalone	0.1154	**0.1688**	0.1384	0.1348
Auction Verification	0.5497	0.5893	0.6667	**0.8243**
Bank Marketing	0.5310	0.7420	**0.7894**	0.7484
Breast Cancer	**0.9634**	0.9502	0.7013	0.9458
Census Income	**0.7328**	0.6664	0.5778	0.6216
Credit Approval	0.5069	0.5176	0.5130	**0.5391**
Heart Disease	0.3244	0.3278	**0.4013**	0.3545
Soybean Disease	0.3986	0.4075	0.3292	**0.4413**

Figure 5.5: Comparison of Accuracy of various Autoencoder architectures combined with k-means on clustering mixed-type datasets.

	All cols AE	AE with Attention	Transformer N=6
Abalone	0.1603	**0.1681**	0.1612
Auction Verification	**0.1234**	0.0109	0.0045
Bank Marketing	0.0040	**0.0394**	0.0022
Breast Cancer	0.5521	0.1906	**0.5569**
Census Income	**0.1475**	0.0237	0.0056
Credit Approval	0.0123	0.1833	**0.3061**
Heart Disease	0.1550	0.1244	**0.1863**
Soybean Disease	**0.5656**	0.2514	0.3966

Figure 5.6: Comparison of Normalized Mutual Information between an autoencoder, an autoencoder using the attention mechanism and a 6-layer Transformer with an autoencoder on clustering mixed-type datasets using k-means.

	All cols AE	AE with Attention	Transformer N=6
Abalone	0.1503	**0.1575**	0.1345
Auction Verification	**0.8331**	0.6422	0.6393
Bank Marketing	**0.7416**	0.7044	0.5636
Breast Cancer	**0.9136**	0.7277	0.8799
Census Income	0.6740	**0.7146**	0.6812
Credit Approval	0.5360	0.7335	**0.7887**
Heart Disease	0.4448	0.3411	**0.4515**
Soybean Disease	**0.4751**	0.2473	0.3114

Figure 5.7: Comparison of Accuracy between an autoencoder, an autoencoder using the attention mechanism and a 6-layer Transformer with an autoencoder on clustering mixed-type datasets using k-means.

Chapter 6

Conclusion

Bibliography

[1] Amir Ahmad and Shehroz S. Khan. Survey of state-of-the-art mixed data clustering algorithms. *IEEE Access*, 7:31883–31902, 2019.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[4] Dana H. Ballard. Modular learning in neural networks. In *AAAI Conference on Artificial Intelligence*, 1987.

[5] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., USA, 1995.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[7] Peter Bloem. Transformers from scratch. `https://peterbloem.nl/blog/transformers`, August 2018. Accessed: 2023-08-24.

[8] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 503:92–108, 2022.

[9] J. C. Gower. A general coefficient of similarity and some of its properties. *Biometrics*, 27(4):857–871, 1971.

[10] Xin Huang, Ashish Khetan, Milan Cvitkovic, and Zohar Karnin. Tabtransformer: Tabular data modeling using contextual embeddings, 2020.

[11] Zhexue Huang. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data Mining and Knowledge Discovery*, 2:283–304, 1998.

[12] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., USA, 1988.

[13] Andras Janosi, William Steinbrunn, Matthias Pfisterer, and Robert Detrano. Heart Disease. UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C52P4X.

[14] Ron Kohavi. Census Income. UCI Machine Learning Repository, 1996. DOI: https://doi.org/10.24432/C5GP7S.

[15] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[16] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[17] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The planar k-means problem is np-hard. *Theoretical Computer Science*, 442:13–21, 2012. Special Issue on the Workshop on Algorithms and Computation (WALCOM 2009).

[18] Kolby Nottingham Markelle Kelly, Rachel Longjohn. The UCI Machine Learning Repository. `https://archive.ics.uci.edu`. Accessed: 2023-07-16.

[19] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[20] S. Moro, P. Rita, and P. Cortez. Bank Marketing. UCI Machine Learning Repository, 2012. DOI: https://doi.org/10.24432/C5K306.

[21] Warwick Nash, Tracy Sellers, Simon Talbot, Andrew Cawthorn, and Wes Ford. Abalone. UCI Machine Learning Repository, 1995. DOI: https://doi.org/10.24432/C55C7W.

[22] Elaheh Ordoni, Jakob Bach, Ann-Katrin Fleck, and Jakob Bach. Auction Verification. UCI Machine Learning Repository, 2022. DOI: https://doi.org/10.24432/C52K6N.

[23] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[25] G. Philip and B. S. Ottaway. Mixed data cluster analysis: An illustration using cypriot hooked-tang weapons. *Archaeometry*, 25(2):119–133, 1983.

[26] Quinlan Quinlan. Credit Approval. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5FS30.

[27] F. Rosenblatt. The perceptron - a perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, Ithaca, New York, January 1957.

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[30] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[31] William Wolberg. Breast Cancer Wisconsin (Original). UCI Machine Learning Repository, 1992. DOI: https://doi.org/10.24432/C5HP4Z.