DeepSqueeze: Deep Semantic Compression for Tabular Data

Amir Ilkhechi Andrew Crotty Alex Galakatos
Yicong Mao Grace Fan Xiran Shi Ugur Cetintemel
Department of Computer Science, Brown University
firstname_lastname@brown.edu

ABSTRACT

With the rapid proliferation of large datasets, efficient data compression has become more important than ever. Columnar compression techniques (e.g., dictionary encoding, run-length encoding, delta encoding) have proved highly effective for tabular data, but they typically compress individual columns without considering potential relationships among columns, such as functional dependencies and correlations. Semantic compression techniques, on the other hand, are designed to leverage such relationships to store only a subset of the columns necessary to infer the others, but existing approaches cannot effectively identify complex relationships across more than a few columns at a time.

We propose DeepSqueeze, a novel semantic compression framework that can efficiently capture these complex relationships within tabular data by using autoencoders to map tuples to a lower-dimensional representation. DeepSqueeze also supports guaranteed error bounds for lossy compression of numerical data and works in conjunction with common columnar compression formats. Our experimental evaluation uses real-world datasets to demonstrate that DeepSqueeze can achieve over a 4× size reduction compared to state-of-the-art alternatives.

ACM Reference Format:
Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, Ugur Cetintemel. 2020. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3318464.3389734

1 INTRODUCTION

In nearly every domain, datasets continue to grow in both size and complexity, driven primarily by a "log everything" mentality with the assumption that the data will be useful or necessary at some time in the future. Applications that produce these enormous datasets range from centralized (e.g., scientific instruments) to widely distributed (e.g., telemetry data from autonomous vehicles, geolocation coordinates from mobile phones). In many cases, long-term archival for such applications is crucial, whether for analysis or compliance purposes.

For example, current Tesla vehicles have a huge number of on-board sensors, including cameras, ultrasonic sensors, and radar, all of which produce high-resolution readings [11]. Data from every vehicle is sent to the cloud and stored indefinitely [30], a practice that has proved useful for a variety of technical and legal reasons. In 2014, for instance, Tesla patched an engine overheating issue after identifying the problem by analyzing archived sensor readings [30].

However, the cost of data storage and transmission has not kept pace with this trend, and effective compression algorithms have become more important than ever. For tabular data, columnar compression techniques [12] (e.g., dictionary encoding, run-length encoding, delta encoding) can often achieve much greater size reductions than general-purpose tools (e.g., gzip [9], bzip2 [4]). Yet, traditional approaches that operate on individual columns miss an important insight: inherent relationships among columns frequently exist in real-world datasets.

Semantic compression, therefore, attempts to leverage these relationships in order to store the minimum number of columns needed to accurately reconstruct all values of entire tuples. A classic example is to retain only the zip code of a mailing address while discarding the city and state, since the former can be used to uniquely recover the latter two. Existing semantic compression approaches [14, 22, 25, 26, 31], though, can only capture simple associations (e.g., functional dependencies, pairwise correlations), thereby failing to identify complex relationships across more than a few columns at a time. Moreover, some require extensive manual tuning [22], while others work only for certain data types [31].

To overcome these limitations, we present DeepSqueeze, a new semantic compression framework for tabular data that uses autoencoders to map tuples to a lower-dimensional space. We additionally introduce several enhancements to the basic model, as well as columnar compression techniques that we have adapted for our approach. As we show in our evaluation, DeepSqueeze can outperform existing approaches by over 4× on several real-world datasets.

In summary, we make the following contributions:

- We present DeepSqueeze, a new semantic compression framework that uses autoencoders to capture complex relationships among both categorical and numerical columns in tabular data.
- We propose several optimizations, including extensions to the basic model (e.g., automatic hyperparameter tuning) and columnar compression techniques specifically tailored to our approach.
- Our experimental results show that DeepSqueeze can outperform state-of-the-art semantic compression techniques by over 4×.

The remainder of this paper is organized as follows. Section 2 provides background on a wide range of compression techniques, including the most closely related work. Next, in Section 3, we present DeepSqueeze and give a high-level overview of our approach. Then, Sections 4-6 describe each stage of DeepSqueeze's compression pipeline in detail. Finally, we present our experimental evaluation in Section 7 and conclude in Section 8.

2 RELATED WORK

Compression is a well-studied problem, both in the context of data management and in other areas. This interest has spawned a huge variety of techniques, ranging from general-purpose algorithms to more specialized approaches.

We note that many of these techniques are not mutually exclusive and can often be combined. For example, many existing semantic compression algorithms [14, 25, 26] apply other general-purpose compression techniques to further reduce their output size. Similarly, DeepSqueeze combines ideas from semantic and deep compression methods, as well as further incorporating columnar (and even some general-purpose) compression techniques.

In the following, we give an overview of the compression landscape and outline the key ideas behind each category.

2.1 General-Purpose Compression

General-purpose compression algorithms are oblivious to the high-level semantics of datasets, simply operating on raw bits. These techniques fall into two categories: (1) lossless and (2) lossy.

2.1.1 Lossless. As the name suggests, lossless compression is reversible, meaning that the original input can be perfectly recovered from the compressed format. Lossless compression algorithms typically operate by identifying and removing statistical redundancy in the input data. For example, Huffman coding [24] replaces symbols with variable-length codes, assigning shorter codes to more frequent symbols such that they require less space.

One common family of lossless compression algorithms includes LZ77 [43], LZ78 [44], LZSS [36], and LZW [42], among others. These approaches work in a streaming fashion to replace sequences with dictionary codes built over a sliding window.

Many hybrid algorithms exist that utilize multiple lossless compression techniques in conjunction. For example, DEFLATE [20] combines Huffman coding and LZSS, whereas Zstandard [16] uses a combination of LZ77, Finite State Entropy coding, and Huffman coding. A number of common tools (e.g., gzip [9], bzip2 [4]) also implement hybrid variations of these core techniques.

2.1.2 Lossy. Unlike lossless compression, lossy compression reduces data size by discarding nonessential information, such as truncating the least significant digits of a measurement or subsampling an image. Due to the loss of information, the process is not reversible. While some use cases require perfect recoverability of the input (e.g., banking transactions), our work focuses on data archival scenarios that can usually tolerate varying degrees of lossiness.

The discrete cosine transform [13] (DCT), which represents data as a sum of cosine functions, is the most widely used lossy compression algorithm. For example, DCT is the basis for a number of image (e.g., JPEG), audio (e.g., MP3), and video (e.g., MPEG) compression techniques.

Another simple form of lossy compression is quantization, which involves discretizing continuous values via rounding or bucketing. DeepSqueeze uses a form of quantization to enable lossy compression of numerical columns, which we describe further in Section 4.

2.2 Columnar Compression

The advent of column-store DBMSs (e.g., C-Store[35], Vertica [27]) prompted the investigation of several compression techniques specialized for columnar storage formats, including dictionary encoding, run-length encoding, and delta encoding [12, 45]. While some of these techniques attempt to balance data size with processing overhead (i.e., lightweight compression), our data archival use case focuses instead on minimizing overall data size.

Parquet [2] is a popular column-oriented data storage format from the Apache Hadoop [1] ecosystem that implements many of the most common columnar and general-purpose compression techniques. Although widely used as a storage format for many big data processing frameworks (e.g., Apache Spark [3]), Parquet also has a standalone library, which we integrate as part of DeepSqueeze.

2.3 Semantic Compression

Semantic compression algorithms attempt to leverage high-level relationships in datasets. One of the first semantic compression works [25] proposed an approach based on "fascicles," which represent clusters of tuples with similar column values that can be extracted and stored only once. Similarly, ItCompress [26] uses an iterative clustering algorithm, such that only the differences between tuples and their respective cluster representative must be stored.

Recent work [41] on the minimum description length [32] (MDL) is closely related to these approaches, although with a broader focus on pattern mining. These patterns, however, can be used for effective compression. For example, Pack [37] is an MDL-based compression approach that uses decision trees to determine compact encodings.

On the other hand, Spartan [14] leverages dependencies and correlations among columns, storing only the subset of columns necessary to accurately infer the others. To reconstruct the discarded columns, Spartan uses models based on classification and regression trees to predict the column values. Similar techniques [31], which leverage data skew with variable-length encoding, also attempt to capture correlations through the use of multi-column codes or special sort orders.

Finally, Squish [22] combines Bayesian networks with arithmetic coding to capture a variety of columnar relationships, including correlations and functional dependencies. While the application of arithmetic coding to Bayesian networks has been previously explored [19], Squish extends this approach by additionally considering numerical data types and lossiness. However, Squish can only guarantee near-optimality under strict conditions when the dataset: (1) contains only categorical columns; and (2) can be efficiently described with a Bayesian network.

2.4 Deep Compression

With the unique ability to model highly complex relationships in large datasets, deep learning is a natural candidate for compressing data. For example, convolutional neural networks (CNNs) have been successfully applied to the problem of image compression [29], as well as other related problems (e.g., denoising [40]).

Autoencoders are a type of artificial neural network that can learn to map input data to (typically smaller) codes. As an unsupervised learning algorithm, they have proved particularly useful for tasks like dimensionality reduction [23]. Like CNNs, autoencoders have also been used for image compression [38, 39].

However, to the best of our knowledge, DeepSqueeze is the first semantic compression method that applies autoencoders to tabular data. Importantly, DeepSqueeze can capture complex relationships across both categorical and numerical columns, and we incorporate lossiness for numerical values into the model through user-specified error thresholds. Additionally, we propose several optimizations that extend the basic model and adapt columnar compression techniques for our approach. Sections 4-6 describe these key distinguishing features in greater detail.

3 DEEPSQUEEZE

As explained in the previous section, existing semantic compression algorithms can capture only simple columnar relationships in tabular data. DeepSqueeze, on the other hand, uses autoencoders to map tuples to a lower-dimensional space, allowing our approach to model complex relationships among many columns. When combined with our specialized columnar compression techniques, DeepSqueeze can produce extremely compact compressed outputs well-suited for long-term data archival.

In particular, we envision two usage scenarios for DeepSqueeze: (1) batch archival for compressing a large static dataset for long-term storage, such as in scientific applications; and (2) streaming archival for compressing a large volume of messages from many clients, such as autonomous vehicles. For the streaming case, the encoder half of the model can even be pushed to the clients to perform compression before transmission, perhaps with periodic retraining to account for shifts in the underlying patterns present in the data.

In this section, we provide a high-level overview of DeepSqueeze's compression and decompression pipelines. Then, we discuss in detail the different stages of the compression pipeline (i.e., preprocessing, model construction, and materialization) in Sections 4-6.

3.1 Compression

Figure 1 illustrates DeepSqueeze's compression pipeline. As input, DeepSqueeze takes a tabular dataset consisting of any combination of categorical and numerical columns, as well as metadata specifying the column types.

The first step (Section 4) is to preprocess this input data into a format upon which the model can operate. Depending on the data type, we apply a variety of well-known transformations (e.g., dictionary encoding), as well as a version of quantization that respects the specified error threshold for numerical columns.

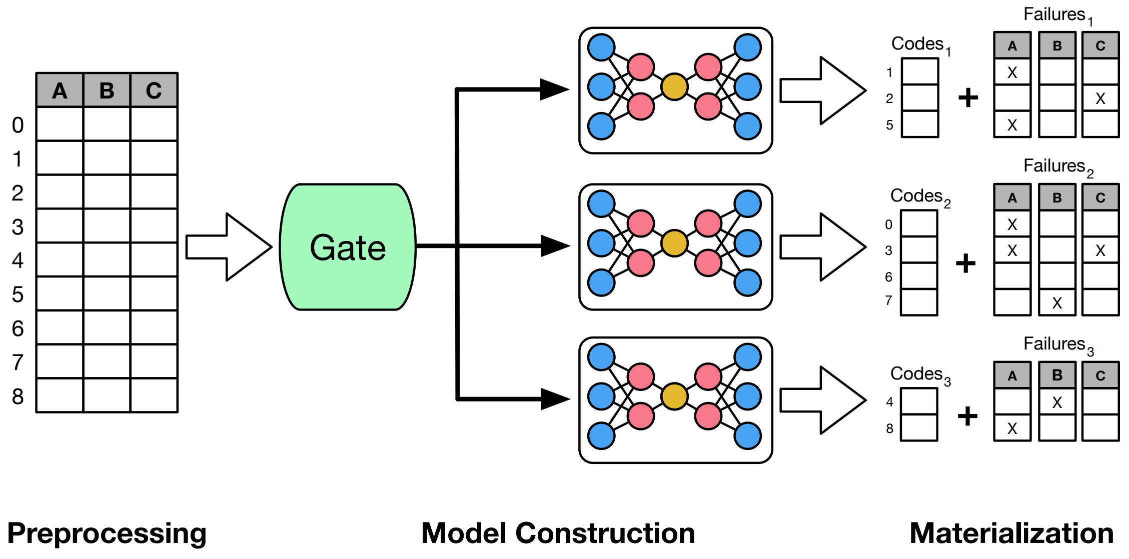


Figure 1: High-level overview of DeepSqueeze's compression pipeline.

In the next step (Section 5), we build a type of artificial neural network called an autoencoder, which maps the preprocessed input data to a lower-dimensional representation. Autoencoders are similar to other dimensionality reduction techniques, but they have the ability to learn complex relationships—which are common in real-world datasets—across many columns at the same time. The unsupervised training process proceeds iteratively over the dataset until convergence, and we have developed several optimization techniques to improve model construction. Importantly, unlike traditional machine learning settings, our goal is to overfit the model to the input data in order to minimize the size of the compressed output.

One way of overfitting the model to the training data is to create a complex model capable of capturing all relationships in the dataset, whereas an alternative approach involves building multiple simpler models, called a mixture of experts, that are specialized for certain partitions of the data that contain similar tuples. Figure 1 shows three specialized models, and tuples are routed to models by a gate that learns the best partitioning of the data during training.

Finally, the materialization step (Section 6) uses the trained autoencoder to generate the lower-dimensional representation, labeled *Codes* in the figure. Each of these codes represents a single tuple and is much smaller in size than the original. We must also save the decoder half of the model in order to reconstruct the original tuples from the codes, as well as failure values to correct mispredictions (denoted with an X) from our model. We propose several extensions to existing columnar compression techniques that we have adapted to work in conjunction with our models.

3.2 Decompression

To decompress the data, we essentially perform the inverse of each step in the compression process. Specifically, the decompression pipeline begins by feeding the materialized codes to the saved decoder to reconstruct an approximate version of the preprocessed input. Then, we compare each reconstructed tuple to the materialized failures and replace any errors with the correct values. After a final step of inverting the preprocessing, we have recovered the original dataset, with potential lossiness in numerical columns that respects the user-specified error thresholds.

4 PREPROCESSING

As mentioned, the first step of DeepSqueeze's compression pipeline is preprocessing, which converts the data to a form appropriate for training our models. This section describes the different preprocessing techniques that we apply to each type of column.

4.1 Categorical Columns

Categorical columns contain distinct, unordered[1] values, usually represented as a string. For columns with few distinct values, dictionary encoding is a well-known compression technique that involves substituting larger data types with smaller codes. DeepSqueeze replaces each distinct value in a categorical column with a unique integer code, which serves two purposes: (1) already reducing the size of the input data; and (2) converting categorical values to a numerical type required by the model. For example, DeepSqueeze would

[1]We do not consider string ordering (e.g., lexicographic order) in this work.

replace the values $\{A, B, C, D\}$ in a column with $\{0, 1, 2, 3\}$, respectively.

On the other hand, for categorical columns with many distinct values (e.g., unique strings, primary keys), DeepSqueeze automatically excludes them from the normal compression pipeline and falls back to existing compression techniques. However, in particular cases when the distribution of the values is skewed, the infrequently occurring values can be ignored during training such that the model is only trained on the most frequently occurring categorical values. Since reducing the number of possible output values for categorical columns can dramatically reduce the number of parameters in the model, the small additional overhead associated with mispredicting infrequent values is offset by the substantial reduction in model size.

4.2 Numerical Columns

Numerical columns can contain either integers or floating-point values. In both cases, the first step is to perform min-max scaling to normalize all values to the $[0, 1]$ range, which prepares the numerical columns for model training and resolves scale differences among column values.

However, many applications can tolerate some degree of imprecision, either because they do not require exact results (e.g., visual data exploration [17, 18, 21]), or because some other form of noise exists in the data generation process (e.g., limitations of sensor hardware). Thus, we also incorporate efficient lossy compression for numerical values by allowing user-specified error thresholds for each column.

One way to permit lossiness is to extend the lossless version by accepting any prediction for a value that falls within the specified error threshold. Additionally, we must modify the associated loss function to account for the error threshold so the model does not penalize these now-correct predictions. While straightforward, this approach still requires us to model a continuous function with a much broader range of possible inputs, and mispredictions are difficult to materialize efficiently because they can have arbitrary precision.

An alternative approach is to quantize the column, which involves replacing the values with the midpoints of disjoint buckets calculated using the specified error threshold. For example, consider a numerical column with values in the range from $[0, 100]$ and a user-specified error threshold of 10%; our quantization approach would replace these continuous values with the discrete bucket midpoints: $\{10, 30, 50, 70, 90\}$. Since quantization already incorporates the error threshold into the bucket creation, we do not need to modify the model or loss function.

Compared to the first approach, quantization allows the model to learn discrete mappings, which has two main benefits: (1) the model can be much simpler and, consequently, smaller in size; and (2) discrete values are easier to predict, resulting in fewer materialized failures. Moreover, columnar compression for the materialized failures of quantized values is much more efficient than continuous values, as described in Section 6. In Section 7.4.1, we show the impact of quantization on the overall performance of DeepSqueeze.

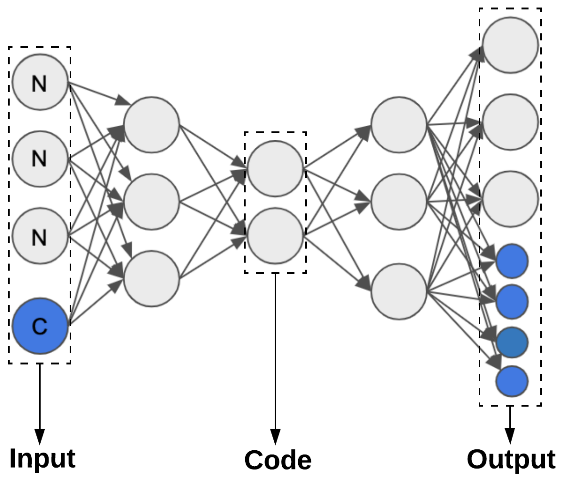


Figure 2: Autoencoder for compressing a dataset with one categorical (C) and three numerical (N) columns.

5 MODEL CONSTRUCTION

This section describes the model construction step of DeepSqueeze's compression pipeline. First, we explain the basic architecture of our model, followed by an extension that uses specialized models for disjoint subsets of the data. Then, we describe our approach for model training and choosing appropriate hyperparameters.

5.1 Basic Architecture

As mentioned, DeepSqueeze uses autoencoders to capture complex relationships among many columns, including both categorical and numerical data types. An autoencoder is an artificial neural network that takes the tuples from the dataset as input, maps them to a lower-dimensional space, and then attempts to reconstruct the original tuples from the lower-dimensional representation. For comparison, a simple autoencoder with only a single hidden layer and no nonlinear activation functions offers modeling capacity similar to principal component analysis techniques for dimensionality reduction. By adding more hidden layers and nonlinear activation functions, we can begin to model increasingly complex relationships.

Figure 2 depicts an example autoencoder for compressing a dataset with one categorical (C) and three numerical (N) columns. As shown, the autoencoder consists of two nearly symmetric models: (1) an *encoder* that maps the input tuples to a lower-dimensional space; and (2) a *decoder* that attempts to reconstruct the input tuples. The shared middle layer represents the learned representation for each tuple, called a *code*. DeepSqueeze always constructs an autoencoder with a smaller representation layer than the original input tuples, which serves as a bottleneck to map the data into a lower-dimensional space. As described in Section 6.2, DeepSqueeze stores these codes and uses the decoder to recreate tuples during decompression.

To handle the mixed categorical and numerical columns often present in real-world datasets, DeepSqueeze needs to dynamically adapt the basic autoencoder architecture based on columnar type information. Notice that the numerical columns in Figure 2 require exactly one node each in the output layer, but categorical columns require one node for each distinct value. In the previous example, a categorical column with possible values $\{A, B, C, D\}$ requires four output nodes (blue in the figure). The outputs for a categorical column produce a probability distribution over the possible values, a fact that we leverage later during the materialization step, which we describe in Section 6.

However, one of the key problems associated with integrating categorical columns is a potential explosion in model size caused by introducing a huge number of connections in the final fully connected layer. To address this problem, we utilize a parameter sharing technique that involves a shared output layer for all categorical columns rather than individual nodes for each distinct value per column. The size of the shared output layer, then, is bounded by the maximum number of distinct values in any categorical column. We must also add an auxiliary layer before the output layer with one node for each categorical column, as well as an additional signal node. The signal node simply provides the index for each categorical column, which informs the shared layer how to interpret the values from the auxiliary layer for a particular output.

Figure 3 depicts the last two layers of a decoder for a dataset with three categorical columns for both the traditional architecture (left) and our version with parameter sharing (right). Each column has a different number of distinct values, which are color-coded. Since the dataset has three columns, our auxiliary layer contains three nodes plus a signal node s. The shared output layer has five nodes, which is the largest number of distinct values across all columns. As shown, the auxiliary layer significantly reduces the size of the fully connected layer.

Finally, the choice of model architecture (i.e., number and sizes of layers) is critical for achieving good performance. While many different heuristics exist, we use two hidden layers, each with twice as many nodes as the number of columns in the input data. We have found empirically that

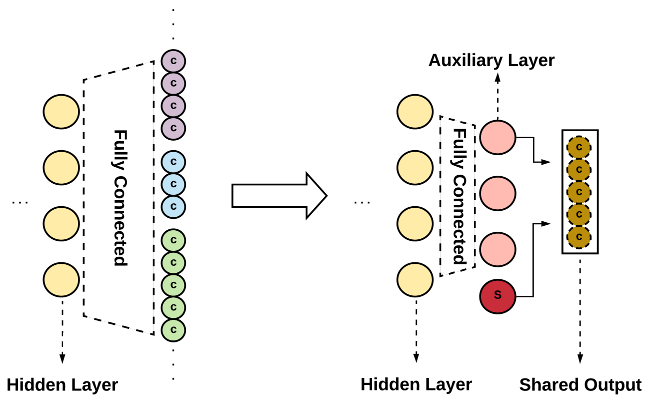


Figure 3: Parameter sharing for categorical columns.

this configuration works well for a wide variety of real-world datasets, such as those tested in our evaluation (Section 7). The number of nodes in the representation layer (i.e., the code size) is a hyperparameter chosen by our tuning algorithm, which we describe later in Section 5.4.

5.2 Mixture of Experts

One straightforward way to improve model accuracy is to increase the size of the network, which in turn increases learning capacity. However, a larger model does not necessarily produce commensurate accuracy improvements, and the increased size might outweigh the compression gains achieved by higher accuracy.

Rather than creating a larger model, an alternative involves building several smaller and less complex models to handle disjoint subsets of the data. The intuition behind this approach is that each smaller model can learn a limited set of simpler mappings.

The obvious choice for partitioning the dataset is to apply a well-known clustering algorithm like k-means, with the idea that tuples grouped into the same cluster will be best represented by the same model. Surprisingly, though, clustering the data might actually end up increasing the required model complexity.

Consider the simple example shown in Figure 4, which compares k-means to our mixture of experts approach for six (x, y) pairs. Using a traditional distance measure, k-means with two centroids would partition the dataset into the two oval-shaped clusters that mix both classes (i.e., gold stars and blue circles). However, note that each of the classes falls roughly along a dotted line, suggesting that they could be easily captured using two simple linear models. Not only are linear models easier to train, but they can also be much more compact and, in this case, will likely produce more accurate predictions.

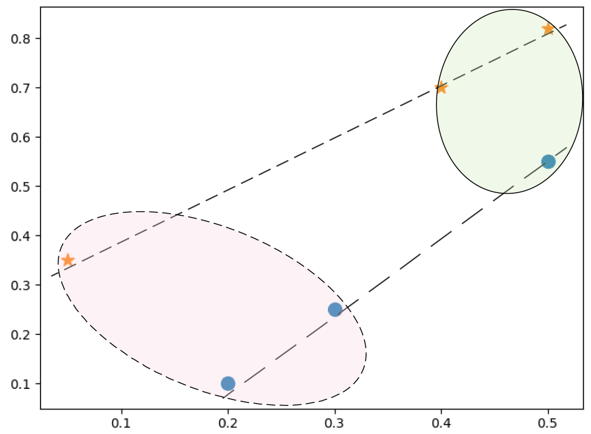


Figure 4: Simple example comparing k-means and mixture of experts.

Based on this observation, we use a sparsely gated mixture of experts [33] that learns how to best partition the dataset in an unsupervised fashion rather than a traditional clustering algorithm. At the core of this architecture is a gate, which is an independent model that assigns tuples to the best-suited expert (i.e., the model with the highest accuracy for each tuple). Therefore, the gate requires one output for each expert.

Figure 1 represents a logical view of the gate, which assigns each of the input tuples to one of the three experts. In practice, though, the input tuples are fed to all experts concurrently, and the gate produces a mask for all but the chosen expert's predictions.

Similar to code size, the number of experts is a hyperparameter that must be chosen on a case-by-case basis. Using too few experts will result in poor model accuracy, while too many experts will needlessly increase the size of the model with no corresponding accuracy improvement. Typically, datasets that are smaller in size or have less variability require fewer experts, whereas larger and complex datasets require more. We explain how our hyperparameter tuning algorithm picks an appropriate number of experts for each dataset in Section 5.4.

5.3 Training

DeepSqueeze uses an end-to-end training procedure in which all parts of the model, including the gate and individual experts, are trained simultaneously. The training process can operate either on the entire dataset or, if the dataset is large, using only a simple. Since a small sample may result in models that fail to generalize well on the full dataset, we show the impact of the chosen sample size on the overall compression ratio in our experimental evaluation (Section 7.4.4). We also explain how our hyperparameter tuning algorithm ensures the use of a sufficiently large sample in Section 5.4.

For training, we repeatedly feed batches of tuples to the model until convergence. The input side of the model takes tuples as-is (i.e., each column requires only a single node, irrespective of type). Recall, however, that the output side requires a special parameter sharing configuration for categorical columns (Section 5.1), where the number of output nodes is equal to the largest number of distinct values across all columns. Therefore, tuples must be reassembled from this one-hot representation after decoding.

For mispredictions, we backpropagate errors to update the weights of the expert responsible for the mispredicted tuple. These backpropagated errors also update the gate, which might choose to reassign the tuple to a different expert.

Importantly, we use different types of loss functions for categorical and numerical columns because, as previously mentioned, their optimization objectives are different. For categorical columns, we use either binary or softmax cross-entropy, depending upon the number of distinct values in the column. Numerical columns, on the other hand, have a closeness property, and we can leverage this fact during the final materialization step (Section 6) to store only the differences between actual and predicted values. Therefore, we want the model to make predictions as close as possible to the actual value, which makes the mean squared error (MSE) a good loss function choice for numerical columns.

5.4 Hyperparameter Tuning

One of the main drawbacks of approaches based on deep learning is the vast number of required hyperparameters. These hyperparameters have a significant impact on model performance and can vary greatly across datasets. As explained throughout this section, DeepSqueeze has several hyperparameters, including the (1) code size, (2) number of experts, and (3) training sample size. Manual tuning of these hyperparameters, however, is often time-consuming and can result in suboptimal models.

Bayesian optimization [34] is an effective search method for finding the best combination of hyperparameters from a range of possible values. Before each trial, an acquisition function predicts the next most promising candidate combination of hyperparameters for the algorithm to try on the next step based on past exploration. Together with the set of candidate values, the algorithm receives a budget that specifies the number of calls to some expensive function that we want to optimize. In our case, this function is the compression part of DeepSqueeze, which performs the model training and compression for the specified raw file. The goal of our hyperparameter tuning procedure is to minimize the overall size of the compressed output.

Figure 5 shows the pseudocode for DeepSqueeze's `tune()` function, which performs the hyperparameter tuning. The parameters include the dataset to compress (`x`), a list of sample sizes to try for training the model (`samples`), and the tolerable error thresholds for each column (`errors`). For the hyperparameters that require tuning, the function also takes `codes`, which is a list of potential output code sizes (i.e., number of nodes in the representation layer), and `experts`, which is a list of the possible number of expert models to try. Finally, `eps` is a user-specified threshold for determining when training has finished.

The tuning function begins with a `for` loop over the provided list of sample sizes. If the candidate sample size `s` is greater than the size of the full data `x`, we can call the `minimize()` function to perform the Bayesian optimization of the hyperparameters and then immediately return the trained model over the whole dataset, as well as the compressed representation. Otherwise, the function proceeds by generating a random sample `x1` from the data of size `s`, which is used as the input to `minimize()`.

The first argument to `minimize()` is the objective function, called `train()`, that returns a trained model `m` and the compressed representation of the sample data `y1`. Again, the goal of the optimization process is to choose the hyperparameter values (i.e., `codes` and `experts`) that will minimize the compressed size. Internally, the `minimize()` function maintains a history of past results for different hyperparameter combinations in order to guide the search toward the most promising candidate hyperparameters.

Next, we generate an independent random sample `x2` (also of size `s`) that is then compressed using the previously trained model `m`, and we compare the sizes of the two compressed outputs in a final cross-validation step. More specifically, we compute the absolute difference in the compressed output sizes normalized by the size of the original data, which gives us a proxy for the generalizability of the model trained with the chosen hyperparameters. If this value is less than the specified `eps` threshold, then we can expect that the model trained on the sample will provide similar performance when applied to the full dataset, and we can therefore return the trained model. Otherwise, the entire `for` loop repeats with a larger sample size.

As mentioned, the `eps` threshold is a user-specified parameter that trades off compressed output size with total runtime. Smaller `eps` values will ensure less variance in model performance (i.e., the model will generalize better), which usually requires a larger sample size and, consequently, longer total hyperparameter tuning time. On the other hand, larger `eps` values will result in much faster runtimes but produce less generalizable models.

Finally, if none of the sample sizes converge, we simply return the model trained using the largest sample.

```
def tune(x, samples, error, codes, experts, eps):
  #iterate over candidate sample sizes
  for s in samples:
    #return model trained on full data if s is too big
    if s >= len(x):
      return minimize(train(x, error), codes, experts)

    #train on sample
    x1 = sample(x, s)
    m, y1 = minimize(train(x1, error), codes, experts)

    #compress separate sample using trained model
    x2 = sample(x, s)
    y2 = m.compress(x2, error)

    #return if size difference within eps
    if abs(y2.size() - y1.size()) / x.size() < eps:
      return m, None

  #return model built on largest sample
  return m, None
```

Figure 5: Iterative Bayesian optimization approach for hyperparameter tuning with increasing sample size.

6 MATERIALIZATION

The final step in DeepSqueeze's compression pipeline is to materialize all of the components necessary for the decompression process, which include the (1) decoder, (2) codes, (3) failures, and (4) expert mappings. The total size of the compressed output, then, is calculated as the sum of each of these individual components. In this section, we describe the different techniques DeepSqueeze applies to further reduce the size of each component.

6.1 Decoder

As previously explained, an autoencoder consists of an encoder that converts a tuple to a compressed code and a decoder that reconstructs (an approximate version of) the original tuple from that compressed code. Since the encoder is required exclusively during the compression process, DeepSqueeze only needs to store the decoder half of the model, which involves simply exporting the weights from each of the experts.

We apply a final gzip compression step on the exported weights to further reduce the decoder size, although this optimization usually provides only a small additional benefit. Other techniques for compressing neural networks are beyond the scope of this work, but they could potentially yield a significant reduction in the size of the model. However, as our experiments demonstrate (Section 7.2), the materialized decoder often represents a relatively small fraction of the overall compressed output size, suggesting that optimization effort would be better spent elsewhere (e.g., reducing the size of materialized failures).

6.2 Codes

Again, the compressed codes produced by the encoder are lower-dimensional representations of each input tuple. Our models produce codes with a minimum width of 64 bits (i.e., a double-precision floating point value), but 64 bits is often unnecessarily large. Therefore, DeepSqueeze iteratively truncates each code until the reduction in code size no longer pays for the corresponding increase in number of failures. Although we could achieve further compression by using variable-length codes, we currently only truncate codes in increments of one byte for simplicity. For several of the datasets in our experiments (Section 7), we were able to decrease the code sizes from 64 to 16 bits, resulting in a 4× size reduction for this component of the compressed output.

Since floating-point values are generally difficult to compress, a final step to convert the codes to integers after the truncation can further reduce the size. For this optimization, we multiply the codes by the smallest power of ten necessary to ensure that the code with the largest number of decimals is converted to a whole number, and then we cast the result to an integer type. The codes can then be compressed more effectively using standard integer compression techniques (e.g., delta encoding).

6.3 Failures

As our experimental evaluation in Section 7 shows, the materialization of failures represents the largest portion of the compressed output size by far. Therefore, we have adapted several traditional columnar compression techniques to minimize failure size, which is much more effective than other semantic compression approaches that apply a final round of general-purpose compression (e.g., gzip). In particular, we use Parquet [2] to compress the materialized failures, and we adapt the standard columnar compression techniques in the following ways for specific column types.

6.3.1 Categorical Columns. For categorical columns, the model outputs a probability distribution over the possible values. Consider again the example categorical column with distinct values $\{A, B, C, D\}$, and suppose our model produces the following probability distribution for a particular tuple: $\{A{=}15\%, B{=}50\%, C{=}5\%, D{=}30\%\}$. One straightforward option is to predict the value with the highest probability (i.e., B), materializing a sentinel value for correct predictions and the actual value for mispredictions. Since most model predictions will hopefully be correct, the repeated sentinel values can be efficiently compressed.

However, if we instead sorted the predictions by decreasing probability, we could store the index of the prediction that matches the correct value. In the example, the value B would be stored as 0, D as 1, A as 3, and C as 4. Assuming the first few predictions of the model are often correct, a variable-length compression scheme (e.g., Huffman coding) could significantly reduce the size.

Binary columns, which have only two possible values, are a special case of categorical columns. Like numerical values, binary values require only a single output node in the model. Rather than using sentinel values to denote correct predictions, we instead encode them as 0 and failures as 1, thereby maintaining a storage size of only a single bit per value. When decompressing, we simply XOR the model predictions with the materialized failures, which will flip values of mispredictions while maintaining the values for correct predictions. Again, while this encoding requires the same number of bits as the original binary column, the goal is to produce long runs of either 0 or 1 values that can then be effectively compressed, using simple run-length encoding or even more advanced techniques [15, 28], when the model makes many correct (or systematically incorrect) predictions.

For example, consider a binary column with eight alternating values: $[1, 0, 1, 0, 1, 0, 1, 0]$. Suppose the model correctly predicts the first five values and mispredicts the final three, in which case our materialization strategy would produce the following output: $[0, 0, 0, 0, 0, 1, 1, 1]$. Compared to the original alternating values, these long runs are much easier for columnar techniques to compress.

Finally, while some semantic compression approaches (e.g., Spartan [14]) support lossy compression for categorical columns by permitting a bounded number of incorrect values in the decompressed output, DeepSqueeze currently only allows lossiness for numerical columns.

6.3.2 Numerical Columns. Unlike categorical columns, numerical columns typically have a much wider range of distinct values. Additionally, the values are ordered, such that some predictions are closer to the actual values than others.

DeepSqueeze leverages the ordered nature of numerical values by storing the difference between the predicted and actual values. For correct predictions, DeepSqueeze will store the value 0, which reflects the fact that the prediction matches the actual value. Again, the intuition is that the predicted values will be close to the actual values, which should significantly reduce the range of values that need to be stored. This approach can even handle systematic errors (e.g., model predictions that are frequently below the actual value by a fixed amount), since the results will still be amenable to columnar compression techniques.

6.4 Expert Mapping

For models with a single expert, we use a single decoder to decompress all tuples. However, for models with multiple experts, we also need to materialize the metadata that maps codes to the correct decoder.

We consider two ways of storing these mappings. The first approach is depicted in Figure 1, where the codes and failures are grouped by expert and stored along with their original indexes. These indexes tell DeepSqueeze the order in which to reconstruct the original file, and they can often be compressed efficiently via delta encoding. The second approach involves storing all tuples together with an additional expert assignment label for each tuple, which DeepSqueeze can then use to select the correct decoder. Similar to storing indexes, these labels can usually be efficiently compressed using run-length encoding. The choice between these two alternatives is data-dependent and must therefore be made on a case-by-case basis.

In some cases, however, maintaining the exact order of tuples in the original dataset is unnecessary, such as for relational tables. Thus, we can save additional space by using the first approach, which stores tuples grouped by expert, without materializing the indexes.

7 EVALUATION

This section presents our experimental evaluation, which uses several real-world datasets. We compare DeepSqueeze against the state-of-the-art semantic compression framework, Squish [22], in terms of both compression performance and overall runtime. We also include results for gzip [9] and Parquet [2], which perform lossless compression, as additional baselines.

In Section 7.1, we first describe the experimental setup and tested datasets. Then, in Section 7.2, we compare the compression ratio for DeepSqueeze to gzip, Parquet, and Squish using the tested datasets for various error thresholds. Section 7.3 measures the runtime of DeepSqueeze relative to the other approaches. Finally, Section 7.4 presents microbenchmarks that show a baseline comparison and breakdown of our optimizations, as well as a detailed analysis of the proposed mixture of experts, hyperparameter tuning, and sampling techniques.

7.1 Setup

Since the key advantage of DeepSqueeze is the ability to capture complex relationships among columns, we conduct our evaluation using five real-world datasets, summarized in Table 1. The Corel and Forest datasets have been used in the evaluation of prior semantic compression work [14, 22, 26], so we include them as a direct comparison point despite their small size. To evaluate DeepSqueeze's performance on larger datasets, we also include three additional datasets: Census, Monitor, and Criteo. We conducted all experiments on a machine with two Intel Xeon E5-2660 v2 CPUs (2.2GHz, 10 cores, 25MB cache) and 256GB RAM.

Dataset	**Size**		**Columns**	
	Raw	***Tuples***	***Categorical***	***Numerical***
Corel [6]	20MB	68K	-	32
Forest [7]	76MB	581K	45	10
Census [5]	339MB	2.5M	68	-
Monitor [10]	3.3GB	23.4M	-	17
Criteo [8]	277GB	946M	27	13

Table 1: Summary of evaluation datasets.

7.2 Compression Ratio

Given our data archival use case, the primary performance metric that we consider in this work is *compression ratio*, which is defined as the size of the compressed output divided by the size of the original dataset. We show the results for the five real-world datasets in Figure 6, with compression ratio expressed as a percentage (smaller is better); for example, a compression ratio of 50% means that the compressed output is half the size of the original dataset.

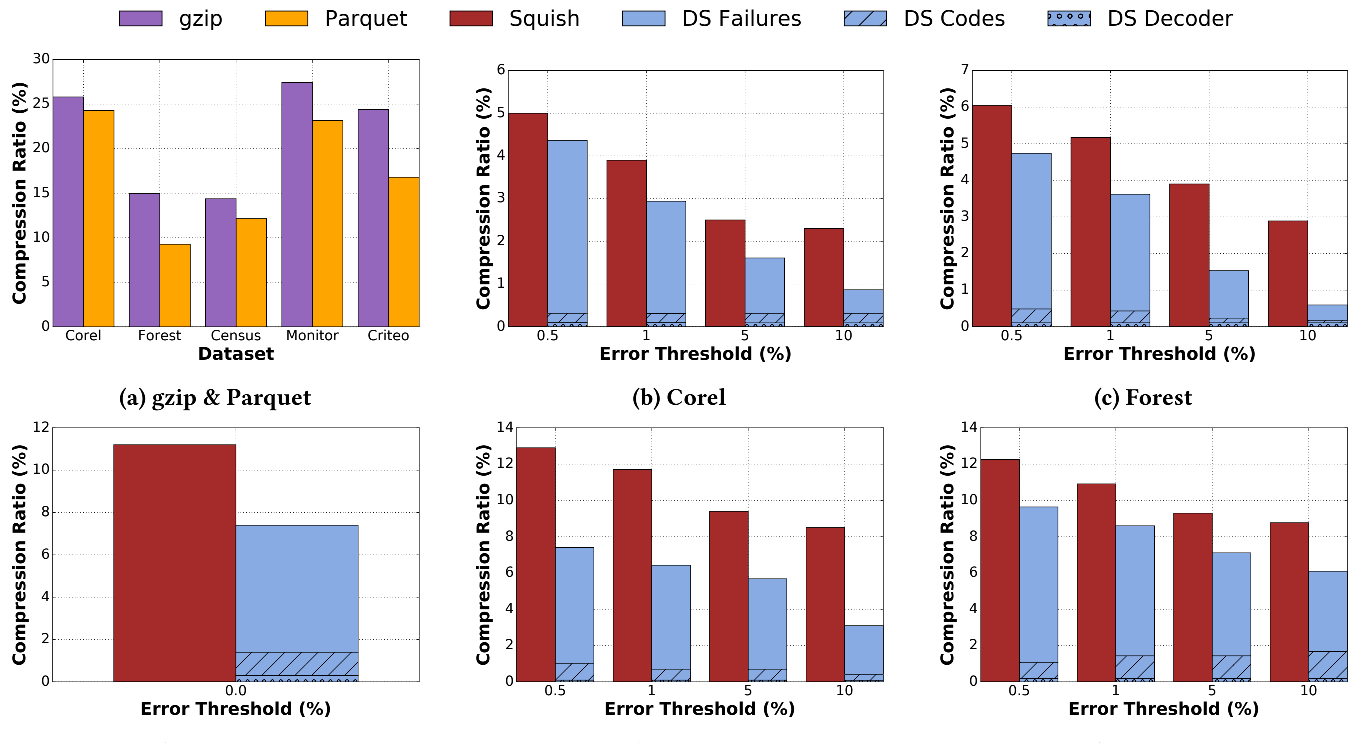
First, as a baseline, Figure 6a shows the compression ratios on each dataset for two lossless compression algorithms: gzip and Parquet. Overall, we see that Parquet generally outperforms gzip on all datasets, ranging from 5% (Corel) up to as much as 37% (Forest).

The remainder of the plots in Figure 6 compare DeepSqueeze's compression ratio to Squish, the state-of-the art semantic compressor. For consistency with prior semantic compression work [14, 22, 26], we report results for the following error thresholds: 0.5%, 1%, 5%, and 10%. As mentioned, the same Corel and Forest datasets were also used in the evaluation of these prior works. Note that, since we use a version of the Census dataset where all numerical columns have been prequantized, Figure 6d contains only a 0% error threshold because neither DeepSqueeze nor Squish permits lossiness for categorical columns. Moreover, this version of Census allows us to test DeepSqueeze on a dataset with purely categorical columns.

Since Squish strongly dominates other semantic compression algorithms (e.g., Spartan [14], ItCompress [26]), we compare only against Squish. In Figure 6, we break down each of the bars for DeepSqueeze into three parts that represent the sizes of the (1) failures and expert mappings, (2) compressed codes, (3) and decoder half of the model.

Overall, we see that DeepSqueeze outperforms Squish across all datasets and error thresholds. In general, for larger error thresholds, DeepSqueeze's models can be simpler and less accurate, leading to fewer failures and a much smaller compression ratio.

For Corel (Figure 6b), DeepSqueeze uses around 25% less space than Squish for an error threshold of 10%. In the same dataset, for an error threshold of 0.5%, the compression ratios



(a) gzip & Parquet (b) Corel (c) Forest

(d) Census (e) Monitor (f) Criteo

Figure 6: Compression ratios for datasets summarized in Table 1.

achieved by the two methods are much closer. The results for the Forest dataset (Figure 6c) show an even greater improvement, with DeepSqueeze outperforming Squish by more than 4×. In terms of dataset characteristics, Census is highly dimensional with low sparsity, whereas Forest is also highly dimensional but with high sparsity. This demonstrates DeepSqueeze's ability to capture complex relationships across many columns.

For the other three datasets, the size reductions achieved by DeepSqueeze are also significant. In Census (Figure 6d), we see that DeepSqueeze offers a 34% improvement over Squish. Similarly, for Monitor (which contains only numerical values), DeepSqueeze outperforms Squish by around 44% with an error threshold of 0.5%, and by more than 63% for an error threshold of 10%. Lastly, for Criteo, we see that DeepSqueeze can still outperform Squish by up to almost 33%, which shows that our techniques can scale to significantly larger datasets.

7.3 Runtime

Although compression ratio is the primary concern for long-term data archival, a reasonable overall runtime is also an important factor. In the following, we compare the runtimes for each approach on the same five datasets with a fixed error threshold of 10%, with results broken down into the time spent on hyperparameter tuning (HT), compression (C), and decompression (D). Table 2 shows the results.

As described in Section 5.4, DeepSqueeze automatically determines appropriate hyperparameters for a given dataset using Bayesian optimization with increasing sample sizes. In Table 2, we see that DeepSqueeze takes between 15 seconds on the smallest dataset (Corel) to just over 1 hour on the largest dataset (Criteo) for hyperparameter tuning. For Monitor and Criteo, our iterative tuning algorithm terminates with a 10% and 0.01% sample, respectively, but uses the full dataset in all other cases.

On all datasets, our hyperparameter tuning process is always on par with or significantly faster than Squish. Also note that the user has complete control over the amount of time that DeepSqueeze spends on hyperparameter tuning, offering the ability to trade off runtime with the overall compression ratio. Moreover, for the streaming usage scenario (Section 3), the cost of hyperparameter tuning is incurred only once up front.

Similarly, DeepSqueeze also offers reasonable runtime performance for both compression and decompression. DeepSqueeze and Squish generally take longer than the other approaches during compression due to the expensive model

Algorithm	Corel			Forest			Census			Monitor			Criteo		
	HT	*C*	*D*	*HT*	*C*	*D*	*HT*	*C*	*D*	*HT*	*C*	*D*	*HT*	*C*	*D*
gzip	-	2	1	-	3	1	-	15	2	-	138	16	-	6,325	618
Parquet	-	3	2	-	21	4	-	113	16	-	296	183	-	7,737	6,392
Squish	15	78	12	128	196	45	6,500	388	322	8,000	954	928	64,800	144,426	140,604
DeepSqueeze	15	6	6	115	12	19	485	128	181	480	487	561	4,307	6,629	13,246

Table 2: Runtimes in seconds for hyperparameter tuning (HT), compression (C), and decompression (D).

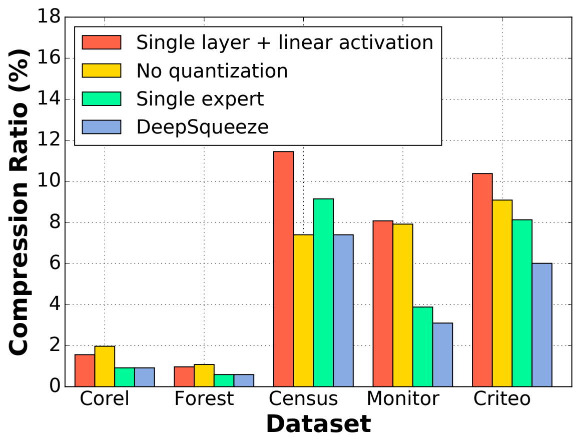


Figure 7: Comparison of different optimizations.

training step. As shown, DeepSqueeze's compression runtimes are reasonably close to that of both gzip and Parquet (within 2×) and, in some cases, even faster (Forest and Criteo). While this result might seem surprising because DeepSqueeze uses Parquet internally to materialize failures, the failures produced by our approach are often easier for Parquet to compress than the original data, since the model will usually correctly predict the output values.

Interestingly, we see that DeepSqueeze is generally faster at compression than decompression, likely due to the fact that our current implementation does not pipeline the tuple reconstruction with writing the output to a file. We plan to implement this optimization in the future.

7.4 Microbenchmarks

This section includes microbenchmarks that evaluate the behavior of DeepSqueeze under different settings. First, we present results that show how DeepSqueeze compares to a simple baseline, as well as the impact of each of our proposed optimizations. Then, we evaluate our mixture of experts approach and hyperparameter tuning algorithm. Unless stated otherwise, all microbenchmarks were conducted with a 10% error threshold.

7.4.1 Optimization Comparison. In order to measure the overall benefits of our proposed techniques, we show the impact of each optimization on all tested datasets in Figure 7, including the quantization of numerical columns and our mixture of experts approach. We compare these techniques against a simple baseline model with only a single layer and linear activations.

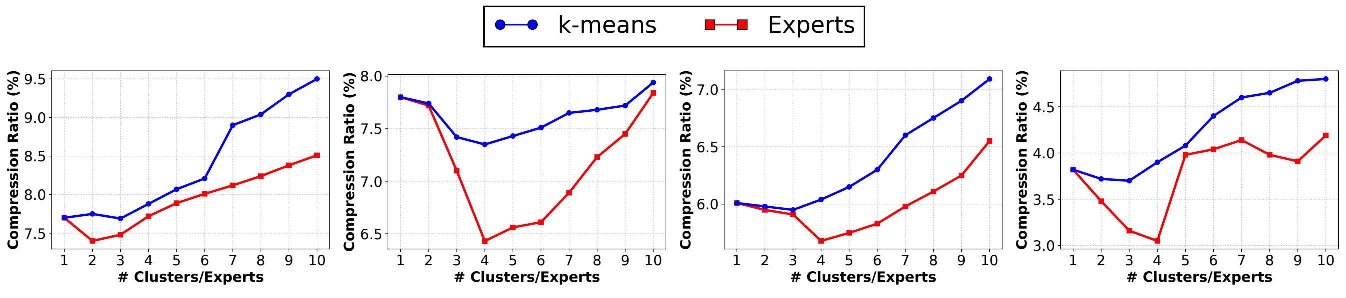
As shown, the baseline model performs significantly worse than DeepSqueeze for all datasets due to limited learning capacity. Similarly, without quantization, the compressed output is considerably larger for all datasets except *Census*, which has no numerical columns. Finally, we see that using multiple experts helps for the larger datasets like Monitor and Criteo, while the impact is not noticeable on the smaller Corel and Forest datasets.

7.4.2 Mixture of Experts. As explained in Section 5, clustering algorithms are a straightforward way of partitioning the data in order to build a set of specialized individual models. Most of these algorithms, however, cluster tuples based on distance measurements rather than statistical or distributional properties. On the other hand, our mixture of experts approach learns to explicitly partition the dataset during the training process.

Figure 8 compares the mixture of experts to k-means for a varying number of clusters/experts for the Monitor dataset. Overall, the mixture of experts approach outperforms k-means across the board. For the largest error threshold (Figure 8d), we see that the lowest compression ratio with no partitioning is around 3.8%. As the number of partitions grows to four, the compression ratio gradually decreases to around 3.1% for the mixture of experts, representing a roughly 18% improvement. On the other hand, the compression ratio actually increases when using k-means, since adding each new model introduces additional storage overhead without improving accuracy. The other error thresholds (Figures 8a-8c) exhibit similar trends.

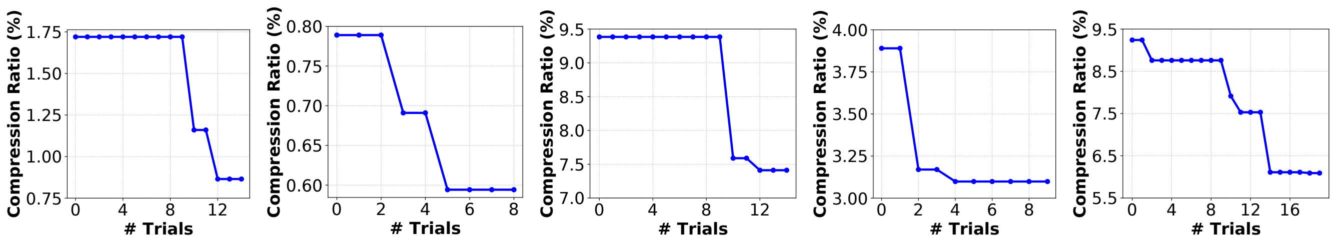
7.4.3 Hyperparameter Tuning. As explained in Section 5, DeepSqueeze's compression model has two main hyperparameters: (1) code size and (2) number of experts. Figure 9 illustrates the number of trials required for the hyperparameter tuning algorithm to converge on each dataset.

Each dataset converges to a different set of hyperparameters. For example, Corel requires a single expert and code size of one, whereas Forest also uses one expert with a code size of



(a) 0.5% (b) 1% (c) 5% (d) 10%

Figure 8: Compression ratios for k-means and mixture of experts with different error thresholds (Monitor).



(a) Corel (b) Forest (c) Census (d) Monitor (e) Criteo

Figure 9: Convergence plots for hyperparameter tuning algorithm.

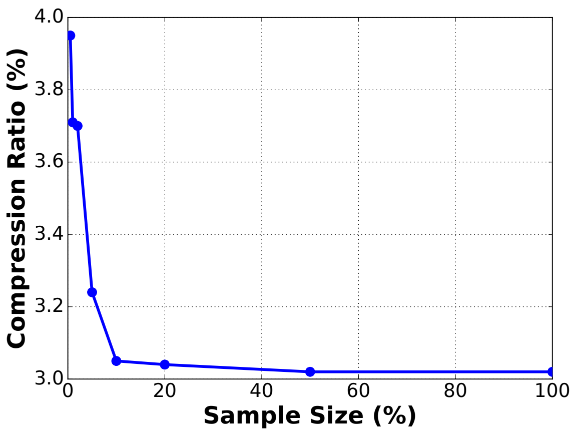


Figure 10: Sensitivity to sample size (Monitor).

two. DeepSqueeze utilizes only a single expert for these files because of their small size, since adding an additional expert increases compressed output size with no improvement in accuracy. For the larger Census, Monitor, and Criteo datasets, our algorithm converges to two, two, and nine experts, with code sizes of two, four, and four, respectively.

7.4.4 Sample Size. For large datasets, DeepSqueeze can use a sample of the data to train the models in order to reduce the overall runtime of the hyperparameter tuning and compression phases. In many cases, training over a sample can offer large performance advantages with minimal loss in generality. However, models that are trained using a small or non-representative sample might not generalize well to the entire dataset, therefore resulting in poor compression performance.

Figure 10 shows the compression ratios for training the model on various sample sizes of the Monitor dataset with an error threshold of 10%. As shown, models trained using sample sizes of less than around 10% are not robust enough to generalize, leading to poor compression ratios. On the other hand, models built using larger sample sizes offer little additional benefit but take significantly longer to train.

8 CONCLUSION

This paper presented DeepSqueeze, a deep semantic compression framework for tabular data. Unlike existing approaches, DeepSqueeze can capture complex relationships among columns through the use of autoencoders. In addition to the basic model, we outlined several optimizations, including automatic hyperparameter tuning and efficient materialization strategies for mispredicted values. Overall, we observed over a 4× reduction in compressed output size compared to state-of-the-art alternatives.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and shepherd for their helpful feedback. This work was funded in part by NSF IIS-1526639 and IIS-1514491.

REFERENCES

[1] [n. d.]. Apache Hadoop. https://hadoop.apache.org/. ([n. d.]).
[2] [n. d.]. Apache Parquet. https://parquet.apache.org/. ([n. d.]).
[3] [n. d.]. Apache Spark. https://spark.apache.org/. ([n. d.]).
[4] [n. d.]. bzip2. http://sourceware.org/bzip2/. ([n. d.]).
[5] [n. d.]. Census. http://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990). ([n. d.]).
[6] [n. d.]. Corel. http://archive.ics.uci.edu/ml/datasets/Corel+Image+Features. ([n. d.]).
[7] [n. d.]. Covtype. http://archive.ics.uci.edu/ml/datasets/covertype. ([n. d.]).
[8] [n. d.]. Criteo. http://labs.criteo.com/2013/12/conversion-logs-dataset/. ([n. d.]).
[9] [n. d.]. gzip. https://www.gnu.org/software/gzip/. ([n. d.]).
[10] [n. d.]. Monitor. https://github.com/crottyan/mgbench. ([n. d.]).
[11] [n. d.]. Tesla Autopilot. https://www.tesla.com/autopilot. ([n. d.]).
[12] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD*. 671–682.
[13] N. Ahmed, T. Natarajan, and K. R. Rao. 1974. Discrete Cosine Transform. *IEEE Trans. Computers* 23, 1 (1974), 90–93.
[14] Shivnath Babu, Minos N. Garofalakis, and Rajeev Rastogi. 2001. SPARTAN: A Model-Based Semantic Compression System for Massive Data Tables. In *SIGMOD*. 283–294.
[15] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2014. Better bitmap performance with Roaring bitmaps. *CoRR* abs/1402.6407 (2014).
[16] Yann Collet and Murray S. Kucherawy. 2018. Zstandard Compression and the application/zstd Media Type. *RFC* 8478 (2018), 1–54.
[17] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2015. Vizdom: Interactive Analytics through Pen and Touch. *PVLDB* 8, 12 (2015), 2024–2027.
[18] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2016. The case for interactive data exploration accelerators (IDEAs). In *HILDA@SIGMOD*.
[19] Scott Davies and Andrew W. Moore. 1999. Bayesian Networks for Lossless Dataset Compression. In *SIGKDD*. 387–391.
[20] Peter Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. *RFC* 1951 (1996), 1–17.
[21] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2017. Revisiting Reuse for Approximate Query Processing. *PVLDB* 10, 10 (2017), 1142–1153.
[22] Yihan Gao and Aditya G. Parameswaran. 2016. Squish: Near-Optimal Compression for Archival of Relational Datasets. In *SIGKDD*. 1575–1584.
[23] Geoffrey E. Hinton and Ruslan Salakhutdinov. 2006. Reducing the Dimensionality of Data with Neural Networks. *Science* 313, 5786 (2006), 504–507.
[24] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.
[25] H. V. Jagadish, J. Madar, and Raymond T. Ng. 1999. Semantic Compression and Pattern Extraction with Fascicles. In *VLDB*. 186–198.
[26] H. V. Jagadish, Raymond T. Ng, Beng Chin Ooi, and Anthony K. H. Tung. 2004. ItCompress: An Iterative Semantic Compression Algorithm. In *ICDE*. 646–657.
[27] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *PVLDB* 5, 12 (2012), 1790–1801.
[28] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi Yan Kai. 2017. Roaring Bitmaps: Implementation of an Optimized Software Library. *CoRR* abs/1709.07821 (2017).
[29] Mu Li, Wangmeng Zuo, Shuhang Gu, Debin Zhao, and David Zhang. 2018. Learning Convolutional Networks for Content-Weighted Image Compression. In *CVPR*. 3214–3223.
[30] Bernard Marr. 2018. The Amazing Ways Tesla Is Using Artificial Intelligence And Big Data. https://www.forbes.com/sites/bernardmarr/2018/01/08/the-amazing-ways-tesla-is-using-artificial-intelligence-and-big-data/. (2018).
[31] Vijayshankar Raman and Garret Swart. 2006. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. In *VLDB*. 858–869.
[32] Jorma Rissanen. 1978. Modeling by shortest data description. *Autom.* 14, 5 (1978), 465–471.
[33] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *ICLR*.
[34] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS*. 2960–2968.
[35] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. 553–564.
[36] James A. Storer and Thomas G. Szymanski. 1982. Data compression via textual substitution. *J. ACM* 29, 4 (1982), 928–951.
[37] Nikolaj Tatti and Jilles Vreeken. 2008. Finding Good Itemsets by Packing Data. In *ICDM*. 588–597.
[38] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. 2017. Lossy Image Compression with Compressive Autoencoders. In *ICLR*.
[39] George Toderici, Sean M. O'Malley, Sung Jin Hwang, Damien Vincent, David Minnen, Shumeet Baluja, Michele Covell, and Rahul Sukthankar. 2016. Variable Rate Image Compression with Recurrent Neural Networks. In *ICLR*.
[40] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. 2018. Deep Image Prior. In *CVPR*. 9446–9454.
[41] Jilles Vreeken. 2009. *Making Pattern Mining Useful.* Ph.D. Dissertation. Utrecht University, Netherlands.
[42] Terry A. Welch. 1984. A Technique for High-Performance Data Compression. *IEEE Computer* 17, 6 (1984), 8–19.
[43] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 3 (1977), 337–343.
[44] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* 24, 5 (1978), 530–536.
[45] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*. 59.